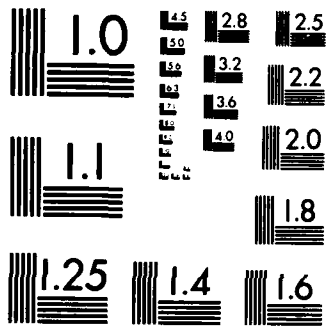
AD-A134 363 MICROCOMPUTER SOFTWARE SYSTEM DEVELOPMENT: SUGGESTED REVISIONS TO MIL-STD..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF SYST.. V M HELBLING 1/2

UNCLASSIFIED SEP 83 AFIT-LSSR-10-83 F/G 5/1 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

MICROCOMPUTER SOFTWARE SYSTEM DEVELOPMENT: SUGGESTED REVISIONS TO MIL-STD-1521A FOR COST-EFFECTIVE ACQUISITION OF CUSTOM SOFTWARE THROUGH SOFTWARE ENGINEERING

Victor M. Helbling, Captain, USAF

LSSR 10-83

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

2

MICROCOMPUTER SOFTWARE SYSTEM DEVELOPMENT: SUGGESTED REVISIONS TO MIL-STD-1521A FOR COST-EFFECTIVE ACQUISITION OF CUSTOM SOFTWARE THROUGH SOFTWARE ENGINEERING

Victor M. Helbling, Captain, USAF

LSSR 10-83

The contents of the document are technically accurate, and no sensitive items, detrimental ideas, or deleterious information are contained therein. Furthermore, the views expressed in the document are those of the author(s) and do not necessarily reflect the views of the School of Systems and Logistics, the Air University, the Air Training Command, the United States Air Force, or the Department of Defense.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER LSSR 10-83	2. GOVT ACCESSION NO. AD-A134363	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MICROCOMPUTER SOFTWARE SYSTEM DEVELOPMENT SUGGESTED REVISIONS TO MIL-STD-1521A FOR COST-EFFECTIVE ACQUISITION OF CUSTOM SOFTWARE THROUGH SOFTWARE ENGINEERING		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Victor M. Helbling, Captain, USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS School of Systems and Logistics Air Force Institute of Technology, WPAFB OH		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Department of Communication AFIT/LSH, WPAFB OH 45433		12. REPORT DATE September 1983 13. NUMBER OF PAGES 104
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES Approved for public release: IAW AFR 190-17. [illegible] LYNN E. WOLAVER Dean for Research and Professional Development Air Force Institute of Technology (ATC) Wright-Patterson AFB OH 45433 15 SEP 1983		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) microcomputer; software acquisition; software engineering; computer system development		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Thesis Chairman: Dr. John A. Muller		

DOD annual investment in computer systems, much of it in microcomputers, will be $38 billion by 1990, up 900 percent from 1980. Software maintenance costs will be 64 percent of the 1990 total, or more than $24 billion. Software maintenance can be greatly reduced through systemic software development as prescribed by MIL-STD-1521A, but DOD managers complain that the process, originally designed for the acquisition of multimillion dollar mainframe systems, not for microcomputers, is much too slow, and therefore not cost effective. Data automation experts point out, however, that development haste in conflict with 1521A increases future maintenance costs. This thesis displays the problem using a recent case study from the Alaskan Air Command, and presents a new acquisition procedure incorporating microcomputer software engineering techniques which reduce system development time while preserving high software quality as intended by the regulations.

LSSR 10-83

MICROCOMPUTER SOFTWARE SYSTEM DEVELOPMENT: SUGGESTED REVISIONS TO MIL-STD-1521A FOR COST-EFFECTIVE ACQUISITION OF CUSTOM SOFTWARE THROUGH SOFTWARE ENGINEERING

A Thesis

Presented to the Faculty of the School of Systems and Logistics

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Systems Management

By

Victor M. Helbling, BGS
Captain, USAF

September 1983

This thesis, written by

Victor M. Helbling

has been accepted by the undersigned on behalf of the faculty of the School of Systems and Logistics in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS MANAGEMENT

DATE: 28 September 1983

[illegible]
COMMITTEE CHAIRMAN

[illegible]
READER

ACKNOWLEDGEMENTS

This thesis could not have been completed without the efforts of many fine people, who materially contributed to its production.

I wish to express my sincere appreciation to my thesis advisor, Dr. John Muller, for his advice, encouragement, support, and friendship throughout the course of the year. I also thank Major Ronald H. Rasch, my thesis reader, for his technical expertise, and Major Ben Iris of the Alaskan Air Command for arranging all the interviews in Anchorage.

One indispensable member of the thesis team was my typist, Jackie McHale, who was solely responsible for this thesis meeting the AFIT style requirements.

Finally, a special thanks to my wife, Rosemarie, for her help in proofreading and rewriting, as well as providing moral and logistical support, and for being my best friend.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The Department of Defense (DOD) procures computers and software the same way it procures airplanes, tanks, and other hardware. The procurement procedure (Appendix A) involves months of work and large quantities of paper. It is designed to ensure that the item finally purchased by the government will serve its intended purpose. This complex and expensive process of acquisition is justified when the object of procurement is itself complex and expensive, but it is not justified, or even useful, in the acquisition of microcomputer hardware and software.

Today a microcomputer costs about the same as a good electric typewriter. Furthermore, the cost of the micro-computer is decreasing every year (41:59-60). The DOD manager will soon be required to procure microcomputers from local vendors, just like office equipment (7), thereby reducing procurement time and cost. Hardware procurement, however, is not the microcomputer's entire life cycle cost. The majority of a computer system's life cycle cost is in software maintenance (15:11). Consequently, the DOD manager, who has the responsibility of minimizing computer system life cycle costs, should concentrate on the cost of software maintenance.

This thesis provides guidelines to minimize the microcomputer's life cycle costs by describing a method of software development which reduces software maintenance cost. These guidelines are intended for the use of all DOD managers; therefore, the thesis speaks to the individual with little knowledge of computers. This thesis acquaints its readers with the concept of software engineering and its use in the procurement of custom software for microcomputers.

Statement of the Problem

To meet increasing demands for information throughout the Department of Defense, managers must depend on computer systems more and more (22:68-72). In the past, these managers were concerned only with mainframe computers and mini-computers, because only these larger machines could handle the large amount of data that needed processing. Recently, due to innovations in technology, the microcomputer has become a suitable alternative. Furthermore, because of aggressive competition among manufacturers, a microcomputer network is cheaper than a single, larger computer. Changes in the procurement procedures of these machines are imminent since the cost of microcomputers is now about the same as that of ordinary office equipment.

These changes will authorize DOD managers to purchase off-the-shelf microcomputers to meet their data processing needs. Industry has already implemented this method of microcomputer procurement (35). It is the DOD manager's responsibility to minimize the life cycle costs of the new microcomputer system. Life cycle costs of computer systems are governed by the amount and ease of software maintenance required by the system. The problem is that, presently, no set instructions are available for the microcomputer user to follow when ordering custom software.

Objectives of the Research

The objective of this thesis is to develop a custom software procurement technique for microcomputers. This technique will be a tool for all DOD managers, and will assume the reader has little or no background in the use of computers. The procedure, presented in Chapter III, is written so any DOD manager/user whose microcomputer requires custom software can ensure that the delivered product will:

1. process the data as intended
2. operate with minimal software maintenance costs, thereby reducing life cycle costs
3. produce the desired information in a reasonable amount of time

4. function in a reliable manner with few interruptions in service caused by logic problems
5. be complete at delivery, including all documentation required to understand and update the software
6. comply with the intent of the regulations (MIL-STD-1521A) by incorporating alternatives which are applicable to microcomputers.

Scope

The microcomputer is a useful tool for DOD managers. The increased capability, availability and reliability of the machine, coupled with its reduced cost, have made investing in a microcomputer cost effective. However, the key factor in the usefulness of a microcomputer lies in the short time from the recognition of the need to the emergence of processed data (information). If obligatory paperwork in the acquisition of hardware and software removes this advantage of time savings, the benefits of technological innovation and economic competition are wasted. Therefore, this thesis assumes that DOD regulations will be modified to delegate the responsibility of the procurement of microcomputer hardware to the DOD manager/user. Furthermore, since the DOD will have recognized the value of time conservation in the procurement of hardware, it is plausible to assume that the same direction will be given regarding software procurement. To fulfill this responsibility, DOD managers, not necessarily experts on software acquisition, require a procedure they can understand to evaluate the development progress of custom software for their microcomputers. This thesis deals only with the management of custom software for microcomputers. It is not intended to guarantee extinction of error, which would not be cost effective. It is intended to minimize life cycle costs in an environment of limited resources (time, expertise, personnel, and money).

It should be noted that hardware complications are not within the scope of this discussion. Therefore, the microcomputer is assumed to be tested and error-free when purchased from the vendor. In addition, it is assumed that DOD managers wish to reduce computer system life cycle costs while increasing the amount of time their system is working properly. Finally, this thesis concerns itself only with custom software procurement, because errors in established software packages have, by-and-large, been eliminated through attrition. Vendors have modernized those portions of their software package which were difficult to use (not "user friendly") as a result of customer complaints. Therefore, "canned programs" will not be addressed in this discussion.

Justification

Between 1970 and 1990, history has and will continue to show a significant worldwide growth in the acquisition of all sizes of computer systems.

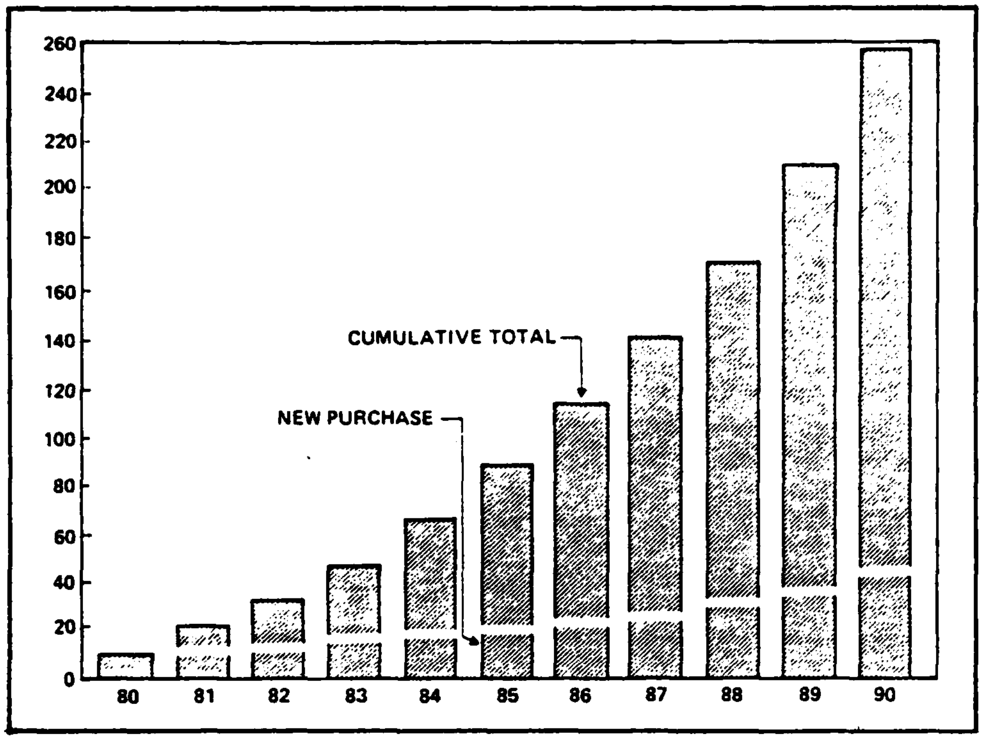


Figure 1. DOD Embedded Computers (15:14)

The DOD investment in computer systems is also increasing at a dramatic rate. As seen in Figure 1, the DOD had less than 10,000 embedded computers in its inventory in 1980-81; by the end of the decade it is expected to have 260,000 (15:13). The embedded computer includes "all computer equipment, programs, data, documentation, personnel and supplies integral to a defense system from design, acquisition or operations and support point of view

[15:19]." The cost of these computers is expected to rise from $4.1 billion in 1980 to $37.99 billion by 1990.

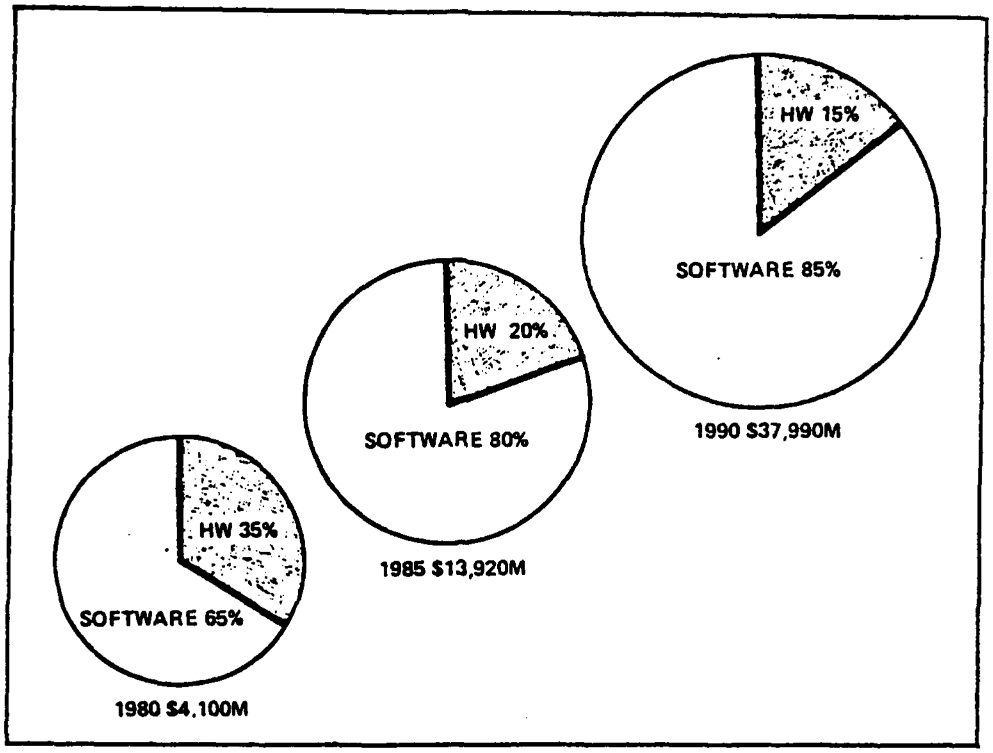


Figure 2. Embedded Computers: Hardware vs. Software (15:14)

As shown in Figure 2, the proportion of the cost attributable to software will increase from 65 percent in 1980 to 85 percent in 1990. This means the software cost will rise from $2.67 billion to $32.29 billion in one decade, an increase of over 1200 percent (15:15). With such a large increase in projected spending, it is imperative that management of software be efficient. Individuals responsible for purchasing software projects should identify in their requirements software formats which minimize life cycle costs. Specifically, the software should be structured so it is easy to maintain. Additionally, the computer system must be a useful tool which justifies its cost. To do that, it must be "user friendly" and reliable, to encourage employees to use it to produce timely information for decision making.

Reliable and "user friendly" systems are not a lucky accident. They result from a systematic approach that remembers the purpose of the devel^opment is the satisfaction of the user. A secondary objective is to reduce the cost of software maintenance.

Software maintenance costs are now estimated to be two-thirds of total computer life cycle cost, which includes hardware, software development and software maintenance. This ratio is expected to worsen (4:73). However, current emphasis is on development costs, while overall life cycle costs continue to be mostly ignored. Numerous experts (including Milne, Rubey, Bunyard, and others) indicate that the cost of software errors increases if they remain unnoticed in the software until the later stages of procurement (26; 39; 4:77). Figure 3 shows that whereas the cost of an error discovered in preliminary design is negligible, the same error detected in the integration phase could cost five times as much. Worse yet, if that error is not noticed until the software is in operation, the maintenance cost is almost 100 times greater (4:77).

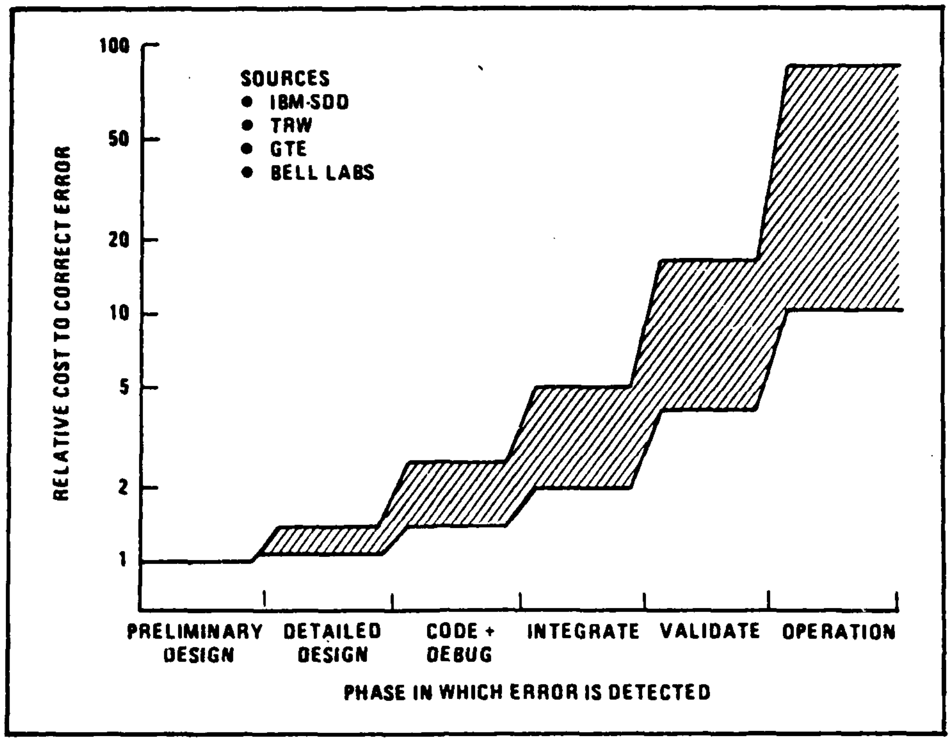


Figure 3. Catching Software Errors (4:77)

When managers procure a software package, they must ensure that the delivered product will transform data into information at the lowest life cycle costs possible. Therefore, the project manager should strive to find any errors embedded in the software at the earliest possible opportunity.

As mentioned earlier, the microcomputer will become readily accessible to every manager in DOD in the near future. However, circumventing the established procedure to save time also bypasses the experience and knowledge accumulated in the local data automation organizations. The procedure established in Chapter III is a "band-aid" for the manager. Although not as specific, it is better because it meets the objectives. It allows the manager to proceed with the operational requirements of the department with a minimum loss of time.

In summary, DOD expenditures on computer systems are growing rapidly. The growth of software maintenance cost is even more rapid. Software cost can be reduced by detecting errors early in the development of the computer system. The cost of software maintenance is a problem to every computer system, including microcomputers. The use of the custom software procurement techniques identified in MIL-STD-1521A (to reduce software maintenance and life cycle costs) in the procurement of microcomputer custom software is time prohibitive. Thus, a new technique is required which considers the intent of the regulations and the time constraints of the user.

Methodology

This thesis develops a low cost method for the reduction of microcomputer system life cycle costs. Initially publications were searched to find established methods that can be used within the DOD. Then users in industry were interviewed to identify any procedures not yet in print. Third, the development of existing systems was studied to determine the problems encountered and the problems that resulted from the procedure used. Finally the intent of the regulations was determined and a new procedure was developed to incorporate the intent of the regulation while allowing for time constraints typical of microcomputers.

Thesis Organization and Overview

This thesis is organized into three chapters and several appendices. Chapter I is the introductory section. It includes an overview, a statement of the problem, the objectives of the research, the scope, the justification, the methodology, and the organization of the thesis. Chapter II describes the current trend in microcomputer system procurement, discusses the problems with present procurement procedures, and explains the purpose of software engineering. Chapter III presents the new procedure for microcomputer custom software acquisition. The appendices provide detailed supplemental information which the reader of this thesis can use in the development of the microcomputer system. To eliminate misunderstanding, all critical words and phases have been defined in the glossary, Appendix B.

CHAPTER II

FUTURE TRENDS IN MICROCOMPUTER SYSTEM PROCUREMENT

Introduction

This chapter presents a logical argument for change in the procurement of microcomputer (micro) systems. First, it justifies the pending changes in micro hardware and software procurement. Then, it discusses the innovative system development of the Alaskan Air Command. Finally, it introduces software engineering and explains why long term costs must be a primary consideration in software procurement.

Trends of Microcomputer Hardware Procurement

The DOD method of purchasing microcomputers will change for three reasons. First, the present method is not cost effective. Just the paperwork intended to prevent waste is several times more expensive than the machine. Second, it frustrates people in critical career fields who must spend their time finding the tools to do a job, rather than doing the job. Finally, the current method often results in an obsolete micro with outdated software, because jobs for microcomputers are typically of short duration in a changing environment. The long, drawn-out process of hardware procurement, followed by a similarly extended process of software procurement can deliver a system not adequate to solve today's problems.

Cost Effectiveness

Today, a project manager must follow a carefully detailed procedure, established by regulation, to purchase a computer. Unfortunately, the same procedure is followed whether a $1,000 micro or a multi-million dollar mainframe is purchased. The procedure, described in Appendix A, carefully ensures that the needs of the organization are met when the equipment is delivered. It evaluates the needs of the user and the direction of the procurement effort several times during the procurement process. This is also very time consuming and results in several problems.

According to Major Ronald H. Rasch, Associate Professor of Accounting and Information Systems for the Air Force Institute of Technology (AFIT), "this policy of extensive planning is very justified for expensive main-frame systems, but not cost effective for micros [35]." The paperwork required by the regulations actually costs much more than the microcomputer itself. Furthermore, the process can easily occupy programmers, engineers and managers for several months. The salary of any one of these people is more than $3,000 per month. Obviously, the method used to prevent waste is much more expensive than any potential waste. In the process, the Air Force dissipates the energy of its technical people who become frustrated, disillusioned, and more difficult (thus, more expensive) to retain.

Retention of Personnel

As shown in Figure 4, computer professionals, including electrical engineers and computer programmers, have a much higher growth need than social need; that is, the prefer to deal with technology rather than with people.

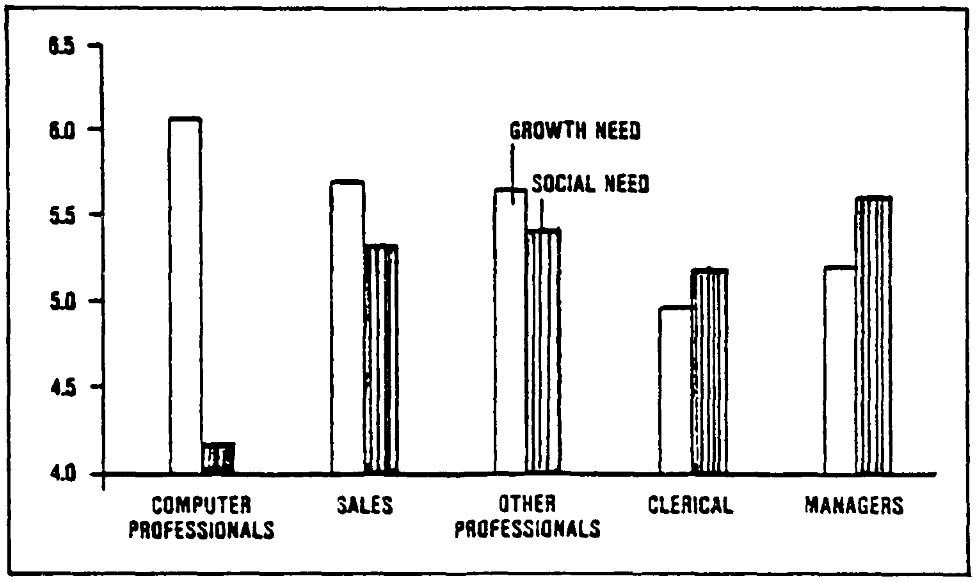


Figure 4. Comparative Needs (39)

Professor Ray Rubey, an expert in software engineering and software acquisition at the Air Force Institute of Technology, stated that the technically oriented employee is not as concerned with monetary compensation as with working on the leading edge of technology. "Studies have verified that scientific programmers will resign from a better paying job to be involved in a project that is clearing new frontiers in the field of computers [39]." That attitude is displayed by astronaut Sally Ride. An August 1983 news report said Ride had been offered several "promotional or advertising offers" worth up to $1 million. The National Aeronautics and Space Administration forbids astronauts to make money endorsing products or appearing in advertising. Ride refused the money. "I wouldn't trade this job for $1 million if they paid me tomorrow [9:2]." With these needs in mind, the Air Force, which is critically short of people in technical fields, should reevaluate the usefulness of the current procurement procedures for micros. An accelerated procurement process will allow scientists to spend less time acquiring tools and more time doing innovative scientific research. The micro is simply an office tool used to process data into a desired format. Although it is more flexible and diverse than a good electric typewriter, the cost is comparable.

Changes in Microcomputer Applications

By the time the process is completed, many changes may have occurred in the requesting organization. Typical changes are in personnel, hardware technology, and system requirements.

According to Major Peter Rensema, who recently managed the installation of the micro-network for the Alaskan Air Command (AAC), the nature of military assignments can result in one of the changes.

> It's not unusual for an individual to order something like a microcomputer and get reassigned. . . . Later, a microcomputer shows up and nobody knows where it came from [12:76].

Although the original purpose of the purchase request can be traced through the process, the individual who conceptualized the idea, the person who best understood the application of the requested system to the problem, may have departed. The probability of the original project being completed is thus significantly reduced.

A second potential change which can occur before the process is complete will be the status of hardware technology. In a year's time, a particular machine can drop from state-of-the-art to more than a generation old. The price and availability of a machine can also change drastically. For example, "the Texas Instrument's 99/4A home computer which sold for $525 when introduced in 1981, retails for just $100 today [41:59]."

Finally, changes in project requirements significantly complicate procurement of the micro system. Until the machine is in hand, there is no limit to the additional capabilities that can be required of it. As requirements increase, the time required to complete the project grows and fewer of the individuals originally involved in the project will remain to complete it. In the meantime, the computer remains unpurchased and the job it was to do remains undone, or the data is manually processed to provide information. These problems can be remedied by procuring micros like any other piece of office equipment.

General Lynnwood E. Clark, Commander of the AAC, expressed his views in an interview June 17, 1983 in Anchorage. He stated that:

> Micros today exceed the capability of many minis and mainframes of a decade ago. They are less expensive and more reliable than the larger computers, as well. Whereas a mini or a mainframe requires a mandatory maintenance capability, a micro provides a redundant capability [failure of one micro does not halt operation of the remainder]. Still, micros must be networked to be as functional as the larger computers [7].

General Clark views the use of micros as the cure to a current data automation problem. As computers become essential to the operation of a variety of departments, data automation specialists are called upon to be familiar with an increasing amount of technical jargon and departmental detail. In his opinion, "we must get the guy who will use the system into the system. That is, we must train each user to be the programmer." Rather than have each programmer become a fuels expert, a supply expert, or a civil engineer, he suggests we train each specialty in the use of the computer, specifically the micro. He continued,

> It is easier and more cost effective. Additionally, it encourages the use of the computer by other members of the organization. The Air Force emphasizes delegation of authority. We must depend on the integrity and professionalism of our people to purchase, set up, and use these tools to improve their response to mission requirements [7].

Alaska Defense Network

General Clark lives his philosophy. The June 1983 issue of Softalk describes how the Alaskan Air Command automated to develop "a state-of-the-art computer system for their alternate command post (ALCOP) that serves as a backup command center for the Alaskan Air Command [12:75]." The system was to have portability, survivability, redundancy, sustainability, as well as be menu driven and user-friendly.

> What General Clark wanted was a computerized database that would travel. One that would still work even if the main terminal was knocked out, or several supporting ones went under. One that anyone commandeered in an emergency could boot and run in thirty minutes. And one that could be updated as disaster progressed, whether it be trembling earth or falling bombs [12:75].

One last requirement which the general identified was that the system had to be on-line by the summer of 1983, just eighteen months from the conception of the project.

Captain Pete Rensema, who was given project responsibility, received a lot of support. "The general saw to it that we only went one week between the paperwork and the first computer [12:76]." Obviously, their acquisition process did not follow the established procedures outlined in the current regulation. According to Captain Dan Rambow, who directed the technical effort as engineering manager, this resulted in a lot of resistance from the Data Automation Department. Data Automation was locked into the use of regulations and established procedures. Fortunately, the General was not.

> They refused to believe you could build an information management system from the bottom up, using personal computers. They suggested a Hewlett-Packard mini with six stations or a mainframe with a multi-user system [12:77].

> What Captain Rambow did was to follow the intent of the regulations (to organize the effort) instead of the letter of the regulations, which was time prohibitive. . . . To meet the letter of the regulations AAC would require significantly more than the 34 man-months used (and available) for the project [13].

The system costs, so far, are much lower with the micros than they would have been with a mini.

> The micro system cost $220,000 for hardware versus $500,000 for the HP, . . . we did it in half the estimated man-months projected using the regulations, and we met our schedule. [12:76] . . . Our software maintenance is quicker and cheaper as well [7].

Captain Rambow manages a construction software house during his off-duty hours. He identified software engineering as the key to the success of the AAC network. He insisted all software be kept simple and user-friendly.

A different view is held by AAC's director of data automation, LTC Dennis W. Howard. "We believe in the evolutionary process versus crash project development. Our concern is minimization of life cycle costs [18]." He added that timeliness will often suffer in order to achieve their primary goal.

> Software maintenance is very expensive due to excessive regulations and documentation requirements. We need to reduce the time required for software maintenance. A standardized language like Ada will help [18].

LTC Howard explained the conflict regarding software engineering between the data automation (DA) staff and the operational staff arises from different approaches to the problem. "The DA people are not prone to [take] risks." They are much more systematic, cautious and tend to 'follow the letter of the regulation'."

General Clark acknowledged that, in some instances, the delays built into Air Force procedures are intolerable to the operational commander. If timeliness is considered the first priority, the command must accept increased life cycle costs of software maintenance.

LTC Howard stated, "The DA shops must not forget they serve the operations staff." Data automation's purpose is to provide the user with information. In the Air Force, the operational staff is the user. Still, LTC Howard advised that:

> A [DA] manager must understand the reason for the regulation and communicate to the user [operational staff] why it is beneficial to them [to follow the established procedures] [18:83].

In AAC's case, the benefit would have been reduced life cycle costs because of minimized software maintenance. The problem the operational staff may encounter is that many of the regulations they failed to comply with require documenting information needed for continuity. LTC Howard wondered about the future:

> Once Major Rensema and Captain Gaudreau have left, who will be the system integrator? Who will maintain the system when Captain Rambow leaves? [18]

The answer, like the problem, revolves around the newness of computers to many users. In ten years most high school graduates will have a good background in the use of computers. A standardized language will probably have evolved. Today, however, we must not allow timidity to impede progress. According to Captain Rambow (1983), "A one-year review for a piece of equipment that has the cost of a typewriter is a waste." In the future, a project manager will not buy a machine specifically designed for his/her purpose. Rather, machines will be selected that meet his/her specifications from the equipment available on the market. The final selection will be based on four machine criteria: 1) capability to efficiently handle the custom software; 2) potential to expand as the needs of the project grow; 3) the overall cost; and 4) reliability.

Once AAC had selected its machine, the engineering manager had to minimize life cycle costs through careful organization and development of the software. It should be noted that the hardware owned by the user presents a constraint for the software engineer. Greentree Computer Corporation quoted an article in Money magazine (November 1982) which suggested that "the first time buyer choose the software first and then buy the compatible hardware [14:1]." Therefore, if you have a need to process data using a computer, acquire the software first, then find a machine compatible with your software. Since the AAC bought the machines and then developed the software, the engineering manager was limited by the machine during software development. In the case of AAC, it did not impose a problem. Their concerns revolved around time constraints, a short-term problem. Captain Rambow's background in software programming and maintenance taught him that the key to minimizing both short-term and long-term problems in software development revolved around a concept known as software engineering.

Software Engineering Background

Awareness that the cost of fixing computer software was spiraling upward spurred the concept of software engineering. One of the first uses of the term 'software engineering' was "in naming the first NATO conference on Software Engineering in 1968 [32:5]." Software engineering is defined as the

> practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them [38:11].

The purpose of software engineering is first, to provide a plan for the "building" of a program in order to increase the probability of building correctly the first time; and second, to facilitate software maintenance.

Until the 1970s the cost of hardware was the major concern of computer facilities managers. Earlier, as depicted in Figure 5, DOD hardware costs constituted over 80 percent of the computer system life cycle costs (15:11). Computers at that time were more magic-boxes than tools.

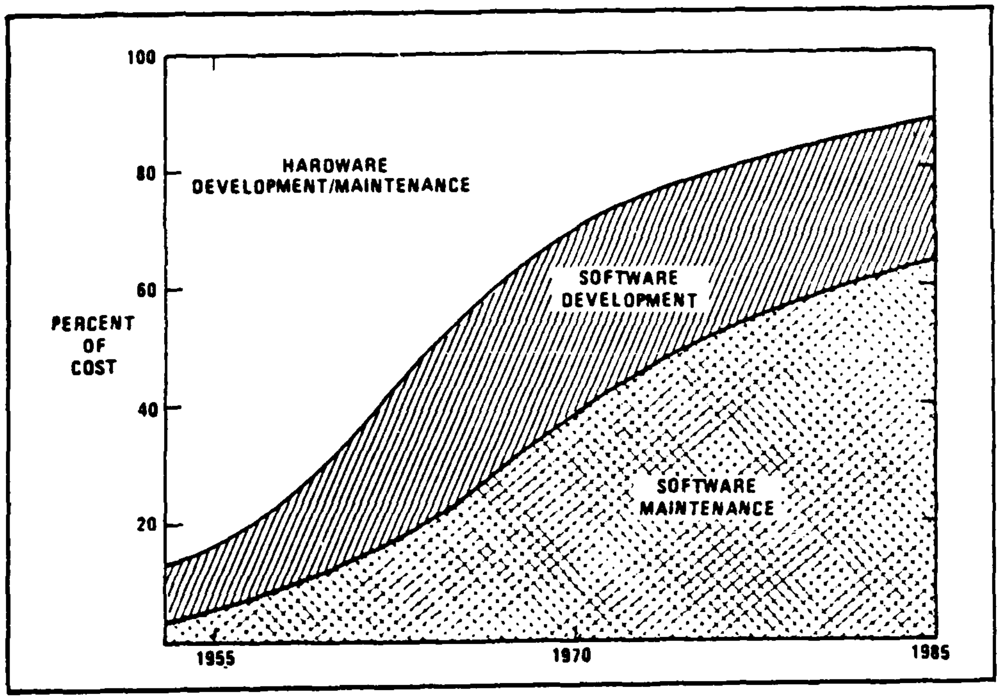


Figure 5. Hardware/Software Cost Trends (4:74)

Computer users were educated people, with backgrounds in mathematics or electrical engineering. Since the computers had limited storage capabilities, programmers had to use the storage efficiently through ingenious strategies when writing computer code, or "software." The useful life of a computer system generation in the 1950s was about ten years and most new machines were "state of the art" for many years. In addition, since hardware costs were too expensive to be cost effective for most firms, few systems existed and programmers had limited job mobility. As a result, managers had the services of their programmers for several years. If a question on the program arose, the software author was available to resolve it. The development of hardware accelerated in the 60s and 70s because of the perceived potential of computers. Now a new generation is developed in less than a year (45).

As depicted in Figure 5, computer system cost trends, which include both hardware and software costs, are changing their apportionment each generation. Software costs exceeding 60 percent of computer life cycle costs today, will constitute over 80 percent of computer system costs by the end of the decade (15:15).

The computer market has expanded as a result of nationwide increases in labor costs and an increasing demand for information from banks of voluminous data. Hardware has become significantly less expensive, primarily because of an expanded market which helped defray the fixed cost of research and development. By 1985, the expected cost of DOD hardware is only 15 percent of the total DOD computer system life cycle cost (15:11).

Software costs are divided into two parts: software development and software maintenance. Software development is the writing of new computer code for a project. As illustrated in Figure 5, the cost of software development has fluctuated between 10 and 20 percent of the DOD computer system life cycle cost. A recent study of nine DOD software projects showed that 50 percent of the software developed is never used. It is the 50 percent that is used that gives rise to the need for software maintenance (39). Software maintenance is updating and correcting software in use to meet the present needs of the organization. Figure 5 shows that maintenance costs have increased during thirty years from 5 percent to over 60 percent of total DOD computer system life cycle cost (15:11). "The GAO recently reported that only 2 percent of software could be used as delivered without changes" (50:51). This increase in maintenance costs results from years of producing software that is more easily updated than rewritten. These updates are generally the result of one of the following:

1. The performance requirements of the system are changed.
2. Hardware which the software must address is modified.
3. Programs with which the software must integrate are revised.

Since the early days of computer programming, the term "program maintenance" has been applied to such updates (11:210). Software maintenance is burdensome because programmers have had no motivation to create clear, complete documentation during the original software development. This forces the maintenance programmer to recreate the logic behind the code, and analyze the thought process of the individual who created the program.

Software maintenance costs can be reduced by meeting the following requirements:

1. Include accurate, clearly written specifications which describe the purpose of the software. The information required from the program should be precisely explained.
2. Perform thorough testing to ensure that the software does what the specifications require.
3. Include clear documentation that explains how the software completes its task.
4. Apply software engineering techniques (described in Chapter III).

Use of these requirements helps ensure that:

1. Program maintenance is performed only for necessary changes.
2. All the capabilities of the system are obvious and therefore useful.

If the documentation is clear and the user understands the function of the software, it is more likely that the system's capabilities will be used before a request for change is made. Any approved changes require software modification followed by software maintenance, which increases system life cycle costs.

Chapter Summary

This chapter explained why the procurement of microcomputer hardware will evolve into a window-shopping event. First, it justified the pending changes in microcomputer purchases of hardware and software. Then it presented the opposing viewpoints in the Alaskan Air Command's innovative microcomputer system development. Finally, it introduced software engineering, and explained the reasons long-term costs must be a primary consideration in software procurement.

CHAPTER III

SOFTWARE ENGINEERING TECHNIQUE FOR MICROCOMPUTERS

Introduction

Software engineering has been identified by experts (39; 26; 32) as a gradual, time consuming ritual intended to satisfy the user's needs through a single effort. It simultaneously attempts to minimize life cycle costs by reducing software maintenance. Unfortunately, the time demanded by established software engineering techniques is not cost-effective when used for microcomputer systems. Obviously, a technique that is not cost-effective is intolerable for use in most organizations. No procedure presently exists, in the military or in industry, for the procurement of custom microcomputer software in a cost-effective manner (24; 28; 45; 36; 1; 52; 16; 8; 23). The procedure developed in this chapter keeps time requirements to a minimum while observing the intent of established software engineering techniques. The procedure is divided into three phases: conceptual, validation, and full-scale development. As in Appendix A, the conceptual phase is considered first.

This chapter assumes some familiarity with the system development methodology. Readers needing additional definitions, descriptions, or detail should either read Appendix A before this chapter, or refer to it as necessary.

Conceptual Phase: Identifying the Operational Requirements

The single most important step in satisfying the user's need is identifying that need (20:20). "One of the most common reasons systems fail is because the definition of systems requirements is inadequate [40:139]." The first step in Peters' waterfall model, "Data Collection," (Figure 6) is a form of system analysis (32:12-13). He defines data collection as the time when "the problem is described, data gathered with which to gauge its magnitude and a fundamental understanding of the problem obtained [32:12]." Identifying the requirements sets the direction of the development effort. The requirements definition must be:

1. technically feasible
2. precise, clear, and not open to misinterpretation
3. produced within the time allotted to this project phase
4. within the project schedules and budgets
5. simple, efficient and economical [40:139]

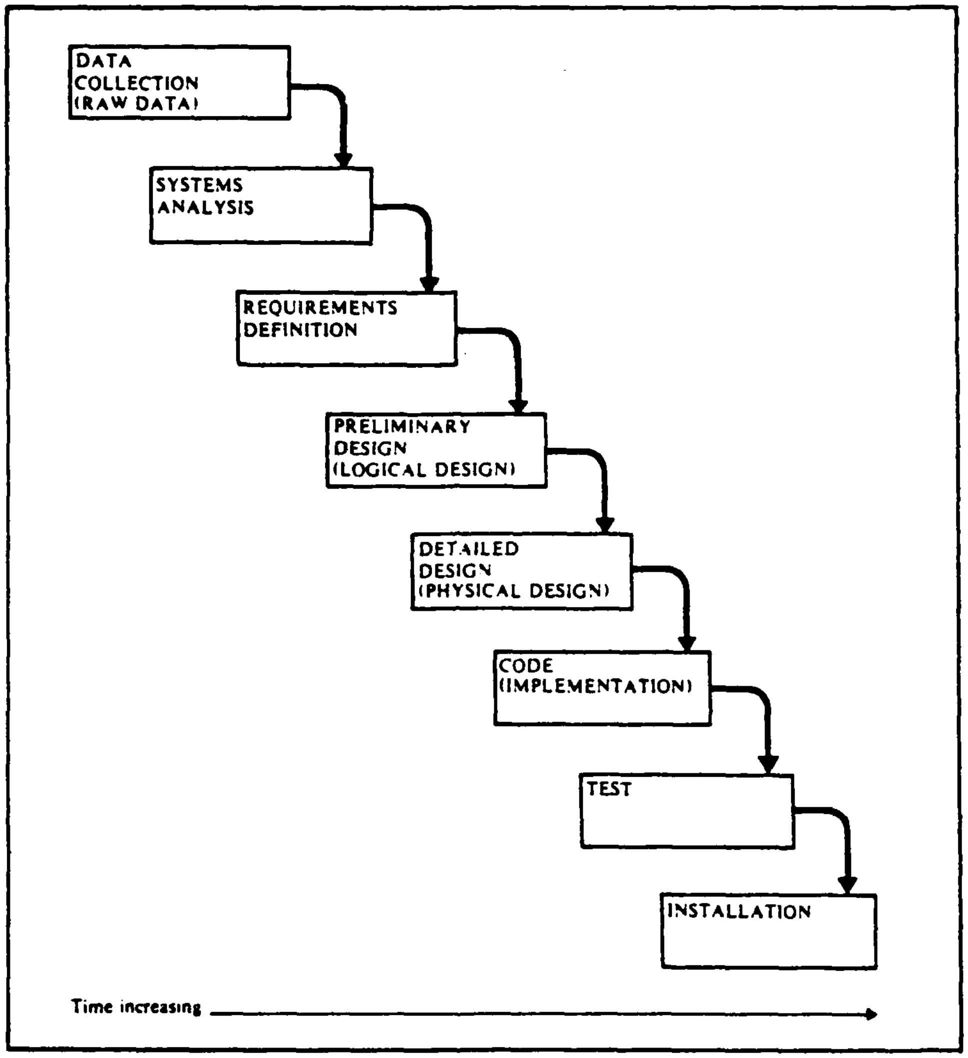

DATA COLLECTION (RAW DATA)
SYSTEMS ANALYSIS
REQUIREMENTS DEFINITION
PRELIMINARY DESIGN (LOGICAL DESIGN)
DETAILED DESIGN (PHYSICAL DESIGN)
CODE (IMPLEMENTATION)
TEST
INSTALLATION
Time increasing

Figure 6. Waterfall Model of the Software Development Cycle (32:13)

If the user can identify exactly what is desired at the beginning of the project, the life cycle costs will be reduced two ways: first, by eliminating the software development firm's costly involvement in requirement identification; and second, by avoiding inappropriate or inefficient software requiring extensive modifications.

The purchase of a microcomputer system is directed toward a specific purpose--transformation of data into information. If the software to perform that transformation is not already available (i.e., a "canned" program does not exist), the DOD manager must pay for expensive custom software development. In this custom development, only the operational requirements which directly satisfy the purpose for which the microcomputer was purchased should be specified. Any additional attributes should be identified as optional. These extras should be considered only if they enhance the operational environment and do not interfere with the primary requirements. Operational requirements are therefore the basis for design, and after the product is complete, the basis for testing. The DOD manager can eliminate ambiguity from operational requirements by "formalizing the user requirements [26]." This formalization includes:

1. a description of the user environment
2. identification of interfaces with other systems
3. identification of operator characteristics
4. a list of functions to be performed
5. a list of inputs to the system and what form the inputs will take
6. a list of outputs from the system and who will use (read) the output. Output should be tailored to meet the need of the user

7. identification of all constraints including

   a. physical (space, power, coolant, machine)

   b. cost (how much you can spend)

   c. schedule (how much time you have)

   d. resources available [26]

(If defining the operational requirements is still a problem the manager can refer to the article by Laura Scharer, "Pinpointing Requirements," pages 139-151, in the April 1981 issue of Datamation.)

After the system's operational requirements are identified they must be recorded. "A problem [requirement] unstated [unwritten] is a problem unsolved [26]." Once these requirements have been recorded they should be communicated to the contractor who will provide the software. The requirements at this time represent the functional baseline (see Appendix A). The manager is now ready to verify the direction of the development as he/she enters the Validation Phase.

The Validation Phase

After the contractor has had the opportunity to study the functional baseline, he/she is ready to discuss these operational requirements and the required techniques of development. Having clearly identified the functional baseline, the next step is to consider reducing life cycle costs. If the DOD manager has been successful in identifying the final functional baseline before initial entry into Full-Scale Development, the rest of the development, including life cycle costs reduction, will be easier and cheaper. Therefore, before the contractor begins design work, the DOD manager should indicate any preference for techniques which aid in the reduction of life cycle software maintenance and system life cycle costs. These techniques are discussed in Appendix C, Formal Program Design Methodologies. The use of these design methodologies will encourage the contractor to carefully plan the software development.

Planning

The planning portion of a project is not only the basis for all other work, but also "requires a third of the entire effort [3:20; 51:198]." Once the contractor understands the operational requirements he/she must prepare a high level design of the proposed system for the DOD manager. The high level design has several benefits which help the engineering effort. The design should:

1. include a graphic representation
2. show the scope of control
3. describe the order of calling
4. show the decomposition
5. identify the inputs and outputs
6. use concise and/or official names

Several techniques are available to help the contractor communicate this high-level design to the DOD manager. These include:

1. Leighton diagrams (32:44-48)
2. Structured Analysis and Design Techniques (32:62-70; 37:16-34; 26)
3. the Systematic Activity Modeling Method (32:136-138; 26)
4. Hierarchy, plus Input, Process, Output (32:48-53; 43:144-154; 26)
5. Data Flow Diagrams (32:139-150; 33:1090; 26)
6. Decision Tables (32:99-100; 42:846; 26)
7. Program Design Language (5:105-110; 32:184-186; 26).
8. Warnier-Orr Diagrams (17:104-174; 32:110,164,176; 26).

> Much as a building architect specifies the structure and construction of a building, the software architect must specify the structure and construction of a program. The major motivation . . . is the desire to reduce the cost of producing and maintaining software (2:13).

At this time the contractor has both the functional baseline (with the operational requirements) and the instructions on how the DOD manager wishes the software development to be organized. These two form the basis by which development cost projections are estimated. Since development contracts are based on this estimate, the importance of the functional baseline and the development instructions cannot be understated. Together, they comprise a preliminary allocated baseline.

A review should be held at this time encompassing the intent of the Systems Requirement Review (SRR) and the System Design Review (SDR) (see Appendix A). This review (the B-5 or Part I Specifications) produces the allocated baseline which is used by the contractor as the minimum standard. The DOD manager and the contractor must at this time agree on the cost and time required to complete the software. The DOD manager should prepare rigid schedules to evaluate the progress of the project. As shown in Figure 5, Appendix A, this ends the Validation Phase and signals the beginning of the Full-Scale Development Phase.

Full-Scale Development Phase

During the first two phases, the DOD manager and the contractor defined the problem and documented the solution. This document, the allocated baseline, is used by the contractor to establish the development guidelines. Now the contractor proposes a custom software design.

To preserve the benefits of the orderly process demanded by regulations and to take advantage of the speed with which microcomputers can be procured, the PDR and CDR should be combined into a single final design review. This composite design review determines if:

1. the design approach considers all performance requirements
2. the approach will satisfy these requirements
3. the design is detailed enough to begin coding.

The manager should carefully analyze the design before accepting it by signing off on the review. He/she should consider exactly how the proposed design would function in anticipated situations. Any deficiencies should be documented because the contractor is responsible for correcting deficiencies before coding starts. The DOD manager should, however, avoid changing the contracted system requirements (the allocated baseline). Any changes to the requirements could affect the time and cost established by the contract. If major changes must be made, the time schedule and the project budget must be adjusted to show a repetition of the Conceptual and Validation Phases. If no changes are necessary the DOD manager should evaluate the quality of the design by using the established measures of software design.

Measures of Software Design

Several measures of good software design have been identified and are applicable in the analysis of micro-computer development. They include coupling, cohesion, and structured programming. The first two measures, coupling and cohesion, evaluate the relationship between and within the software modules of a program. Appendix D discusses coupling and cohesion as measures of good design. The last measure, structured programming,

> is becoming one of the more promising approaches to reducing the ever-increasing cost of producing and maintaining software. The goal of structured programming is to organize and discipline the program design and coding process in order to reduce logic type errors [34:38].

A detailed review of structured programming is provided in Appendix F. Structured programming, like coupling and cohesion, can help the DOD manager evaluate whether or not the software design is adequate (11:212). If it is not adequate, the contractor should be notified of the design problems. The contractor must then correct those deficiencies. Once the deficiencies have been corrected the contractor can translate the design into code. A finished program includes the code and its full documentation (46:208). Full documentation includes: 1) a users' manual 2) a maintenance manual; 3) operational handbooks; and 4) top level design diagrams (25). Once the contractor completes the code, the DOD manager must evaluate its performance before delivery. This evaluation is conducted through software testing (26).

Testing the Software

"Meyers defined testing as the process of executing a program with the intent of finding errors [26]." An error will be present when the program created by the contractor does not do what the DOD manager reasonably expects it to do. In order to find errors the contractor must use a test plan to evaluate the produced software. The test plan should be developed before the testing starts. In current practice, software testing is 40 to 50 percent of the total effort (26; 3:20; 51:199). Still, this effort does not guarantee error free software. "One cannot find all the errors in a program by testing it [19; 46:206; 11:211]." Testing may find some errors, but no reasonable amount of testing can guarantee their absence (26). The three major categories of errors are 1) a mistake in design; 2) a failure of a component; and 3) error introduced by a human operator (27:45). Since costs increase rapidly as an error free system is approached (30), the DOD manager should be concerned only that those errors in the system be eliminated which interfere with the system objectives. Any errors which do not interfere with the operational purpose of the microcomputer often cannot cost effectively be detected or corrected. Correction of detected errors is software maintenance. Unnecessary software maintenance should be avoided. "Development builds a system, maintenance can destroy it [6:199]."

A well planned development is the best way to avoid the first two categories of software errors which are specification related and most common (46:218). Testing will often locate errors in the last two categories. Since the contractor must correct any errors found during testing, the DOD manager should only be concerned that the test plan was followed. Once testing is complete and all corrections made, the contractor has a finished product composed of the software, the documentation, the test plan, and the test results. The DOD manager must now ensure that the completed product is acceptable. This is done by conducting a configuration audit.

The Configuration Audit

The configuration audit of a developed microcomputer software system combines the Functional Configuration Audit (FCA) and the Physical Configuration Audit (PCA) discussed in Appendix A. In this audit the DOD manager determines if

1. the test plan is adequate to ensure that the operational requirements identified in the allocated baseline have been met

2. the test results indicate satisfactory completion of all tests
3. the support documentation accurately and clearly reflects the software (10:1341).

The audit should be performed by a team independent of the development project (48:319). Any discrepancies noted should be documented and corrected by the contractor. Once the contractor corrects the discrepancies, the DOD manager is ready to accept the production baseline of the software with one condition. The software is accepted as delivered, provided the contractor has corrected all discrepancies identified during design review and configuration audit. In addition, any testing which the contractor identified as successful that cannot be verified (by reproduction) is the responsibility of the contractor. The DOD contractor must specify this condition in the Acceptance Contract. The Formal Qualification Review (FQR), which normally is conducted after the audits, is not conducted in the microcomputer custom software procurement process. Now, the DOD manager can accept delivery of the software system, and is ready to purchase the hardware (14:1; 44:II-2). The information required by the DOD manager about the software is provided by the contractor. The DOD manager must consider the constraints imposed by that information as well as the projected growth of the system when he/she selects the hardware. Once the DOD manager has the hardware and the software, this new tool can be used to improve the organization's performance.

Summary

This chapter outlined a procedure which can be used by DOD managers to procure custom microcomputer software. This procedure maintains the intent of the regulations while significantly reducing the time required for the DOD manager to have the microcomputer system working for his/her organization. The procedure, as depicted in Figure 7, has three phases. The conceptual phase identifies the operational requirements which become the functional baseline and are communicated to the contractor. The validation phase confirms the functional baseline and adds the method of development requested by the DOD manager. This results in the allocated baseline. Finally, in the full-scale development phase the allocated baseline is used to design, code, and test the system. Together, these three phases comprise a system which is a compromise between the timely needs of the operational commander and the meticulous development of the data automation officer.

This procedure still requires time. However, it is much more streamlined than a mainframe development (see Appendix A). The time it does consume will not be much

Date	Step
	Conceptual Phase
________	- Identify requirements
________	- Write down requirements = functional baseline
________	- Communicate functional baseline to contractor
	Validation Phase
________	- Identify desired software engineering guidelines
________	- Requirements review
________	- Document/correct discrepancies
________	- Allocated baseline
________	- Contracts and schedules signed
	Full-Scale Development Phase
________	- Contractor proposes design
________	- Analyze design
________	- Conduct design review
	- To verify satisfaction of requirements
	- Measure software design
	- Document/correct discrepancies
________	- Contractor codes the design
________	- Code is tested by contractor
	- Detects errors
	- Corrects errors
	- Retests
	- Documents and tests results
________	- Conduct Configuration Audit
	- Analyze test plan to make certain it verifies that the delivered code meets the requirements of the allocated baseline
	- Evaluate the test results
	- Ensure documentation is complete
	- Document/correct discrepancies
________	- Accept software system
________	- Purchase hardware from local vendor

Figure 7. Microcomputer Software System Development Checklist

more than that required for a "crash" development. In fact, experienced software managers will confirm that the additional software maintenance required with a crash project will make this structured effort more timely (39; 26; 34).

APPENDIX A

AN INTERPRETATION OF MIL-STD-1521A: TECHNICAL REVIEWS AND AUDITS FOR SYSTEMS, EQUIPMENT, AND COMPUTER PROGRAMS

Introduction

This appendix is provided to allow the reader to compare the abbreviated system development methodology, presented in Chapter III, with the existing standard. The difference between the two methodologies is obvious when Figure 7 is compared to Figure 8. The presence of this appendix also allows the reader to review the intent of the original system development methodology. The system development methodology can be used by the DOD manager to reduce software system life cycle costs. This process has seven steps:

1. System Requirement Review (SRR)
2. System Design Review (SDR)
3. Preliminary Design Review (PDR)
4. Critical Design Review (CDR)
5. Functional Configuration Audit (FCA)
6. Physical Configuration Audit (PCA)
7. Formal Qualification Review (FQR)

Using these reviews, the manager monitors the developing organization's technical progress. They also reveal the technical progress of each phase in the system acquisition life cycle. As seen in Figure 9, a life cycle has three phases: 1) the Conceptual Phase; 2) the Validation Phase; and 3) the Full-Scale Development Phase.

Date	Step
	Conceptual Phase
	- Identify requirements
	- Write down requirements
	- Communicate functional baseline to contractor
	- Conduct SRR to determine contractor's initial progress
	- Document deficiencies
	Validation Phase
	- Evaluate software engineering development
	- Conduct SDR for an overall review of the requirements
	- Document/correct discrepancies
	- Allocated baseline
	Full-Scale Development Phase
	- Contractor proposes high-level design
	- Analyze design
	- Conduct PDR to compare high-level design to allocated baseline
	- Evaluate the high-level software design
	- Document discrepancies
	- Accept high-level design *(C-5 specifications)*
	- Initiate detail design of the software
	- Conduct CDR to ensure detailed design satisfies performance requirements of the allocated baseline
	- Evaluate the detailed software design
	- Document discrepancies
	- Contractor codes the design baseline - detects errors - corrects errors - retests - documents test results
	- Conduct FCA - analyze the test plan to make certain it verifies that the delivered code meets the requirements of the allocated baseline - evaluate the test results - document discrepancies in action items
	- Conduct PCA - ensure that the support documentation accurately and clearly reflects the software - make certain the documentation is complete - make certain the software changes resulting from the FCA are reflected in the documentation - document discrepancies in action items
	- Conduct FQR - confirm that all action items identified during the FCA and the PCA have been *resolved*
	- Accept software system
	- Repeat Development steps for the procurement of the hardware

Figure 8. Software System Development Checklist

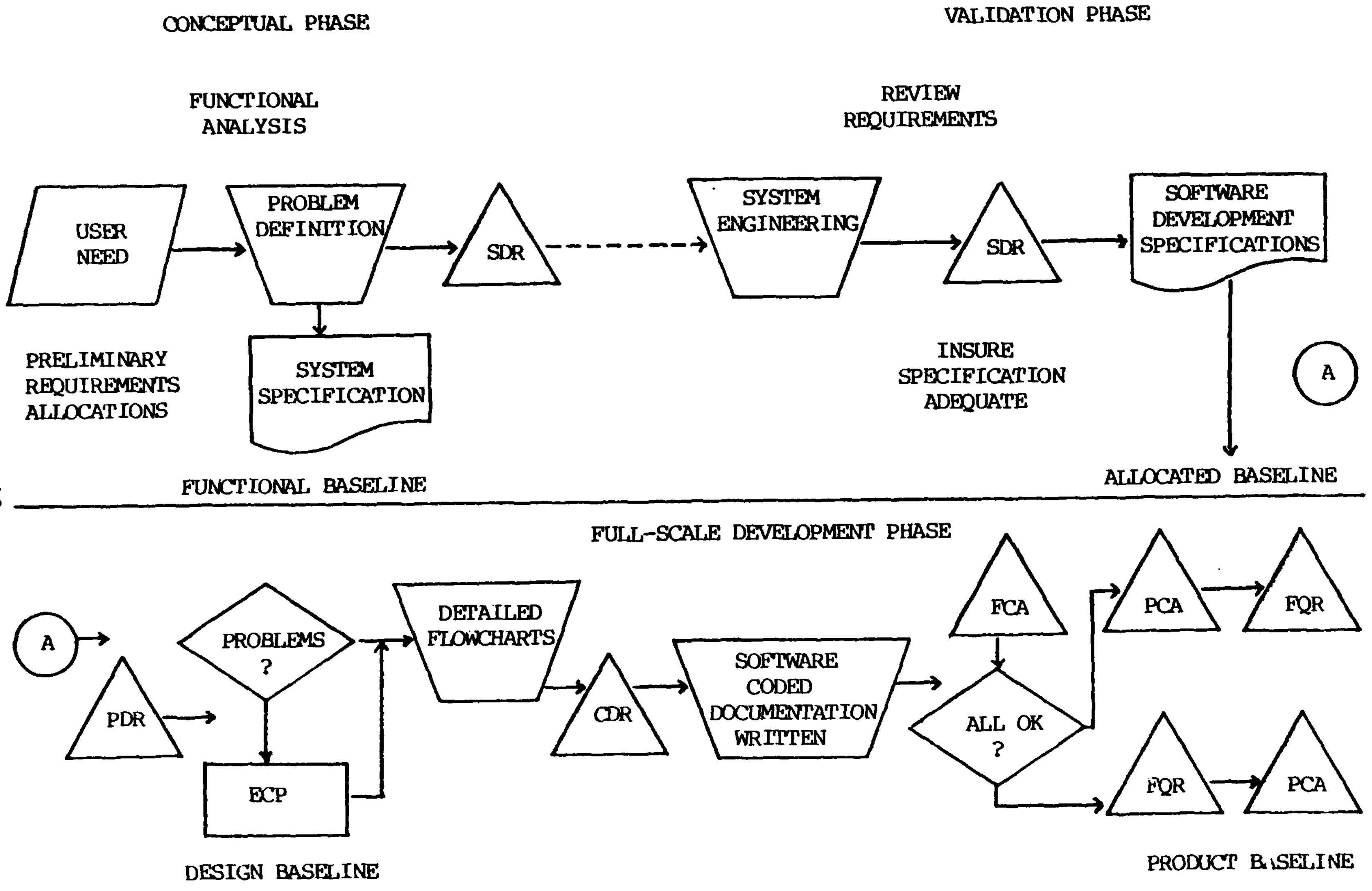


Figure 9. System Development Methodology: Phases, Reviews, and Audits (29:7)

Conceptual Phase

Control of the system life cycle starts with the Conceptual Phase which includes a functional analysis and preliminary requirements allocation. When that is finished the System Requirement Review (SRR) is conducted "to determine initial direction and progress of the contractor's System Engineering Management effort and his convergence upon the optimum and complete configuration [47:12]." The SRR guarantees that the contractor is, indeed, solving the right problem. This phase produces System Specifications, called the functional baseline. The functional baseline identifies what needs to be done. It is a first attempt at describing the specifications. At this point the life cycle enters the Validation Phase.

Validation Phase

During the Validation Phase, the System Specifications functional baseline governs the tasks of the contractor as the product is developed. The Air Force project manager monitors the contractor's progress during the Validation Phase using the System Design Review (SDR). The SDR permits the manager to evaluate the adequacy of the Validation Phase products before the contractor formally submits them (29:20). The SDR is primarily concerned with the overall review of the requirements.

MIL-STD-1521A, Appendix B, lists the items which must be reviewed during the SDR. After the SDR is completed, the contractor should correct any deficiencies identified during the SDR. These authenticated specifications (product) are the B-5 or Part I specification, the allocated baseline. The submittal of the specifications ends the Validation Phase and leads to the beginning of the Full-Scale Development Phase.

Full-Scale Development Phase

Reviews

The first review conducted in the Full-Scale Development Phase is the Preliminary Design Review (PDR). The PDR is a review of the developer's top-level software design in response to the software specifications (29:25). These specifications would have been approved, authenticated and baselined in the phases prior to the PDR. The purpose of the PDR is to determine if 1) the design approach considers all performance requirements; 2) the approach will satisfy these requirements; and 3) the new software will work with the existing software and hardware.

The requirements for conducting PDRs are specified in MIL-STD-1521A, Appendix C (29:26). The PDR is conducted after a basic design approach has been selected by the contractor. If problems in the design of the baseline are discovered during the PDR, the Air Force must ensure that an Engineering Change Proposal (ECP) is issued. The ECP will formally acknowledge the changes to the established baselines (29:30). Upon completion of the PDR, the developer has achieved the design baseline (C-5 specification) and is ready to initiates the detail design of the software. The design baseline signifies the end of the Analysis Phase and the beginning of the Design Phase. When the detail design is completed, and before coding and testing of software, a Critical Design Review (CDR) is conducted. According to MIL-STD-1521A,

> A CDR shall be conducted [on the software] to ensure that design solutions, as reflected in the Draft Part II Product Specifications on engineering drawings, satisfy performance requirements established by the Part I Development Specification [47:40].

The CDR for a software package is a technical review of flowcharts when the logical design is completed (29:30). The purpose of the CDR is to critically review the detailed design of the software to determine if 1) the requirements of the software development specifications can be implemented; 2) the detailed design of the flowcharts is compatible with the design structure presented at the Preliminary Design Review; and 3) the flowcharts are detailed enough to start coding.

To identify deficiencies, the manager should question the design presented. If the deficiencies are extensive and the design is unacceptable, the manager should reschedule the CDR for a later date. Otherwise, if the design is basically sound, the deficiencies should be documented in the meeting minutes of the CDR. This action should be followed up to ensure that the deficiencies have been corrected.

After a successful CDR, the contractor will prepare the software and its documentation. It is then time to conduct the configuration audits on this product.

Audits

The purpose of the audits is to verify compliance with the requirements identified in the specifications and other contract requirements. Two kinds of audits are performed: the Functional Configuration Audit (FCA) and the Physical Configuration Audit (PCA).

The purpose of the FCA is to ensure that the delivered computer code actually does what the user asked for in the Software Specifications (B-S/Part I Specifications). The FCA is accomplished by auditing the results of the software qualification tests to determine the qualification status of the software (29:35). If during the FCA the software fails to meet the specifications, the manager has two choices.

If the discrepancies are not extensive, they should be formally identified in the FCA minutes and in action items. The discrepancies must be corrected before the Physical Configuration Audit starts, and the corrections checked during the Formal Qualification Review.

If the discrepancies are extensive, the software is disapproved, or the requirements in the Development Specification are changed. If the software is disapproved, the software must be modified, retested and re-audited. Sometimes, though, when the software cannot satisfy the Development Specifications, "it may be desirable to change the specifications to agree with the software performance [29:37]." If the software is accepted after the FCA, then the Formal Qualification Review becomes a formality which is part of the FCA. In this case, the FQR, which is essentially the same as the FCA, "can be accomplished in a single combined Audit/Review [47]." If the software has some requirements yet to be satisfied, the FQR will be conducted after the Physical Configuration Audit (29:42).

The Physical Configuration Audit (PCA) is conducted to ensure that the support documentation accurately and clearly reflects the software. It is the prime instrument used for making design modifications (software maintenance) to computer code (29:38). Good complete documentation can significantly reduce software maintenance costs. We know that software maintenance costs comprise two-thirds of future DOD computer life cycle costs (15:11). Therefore, the PCA can help reduce DOD software maintenance costs. The PCA is conducted by comparing the documentation with the software. This includes verifying the narrative information and flowcharts against listings for accuracy, completeness, and understandability of documentation. Special attention must be given to portions of software which were changed as a result of the FCA. It is possible that the corresponding documentation updates may have been overlooked. In addition,

> The positional handbooks, users manuals, and operators manuals should be validated prior to System Development Test and Evaluation (DT&E). The verification of these manuals will normally be accomplished during System DT&E [29:40].

Deficiencies discovered during the PCA will be listed on DD Form 250. This signed DD 250 (with deficiencies) represents a conditional acceptance until the shortages have been satisfied. This conditional acceptance, with outstanding qualification requirements, normally requires a Formal Qualification Review (29:42).

The FQR then ensures that all discrepancies noted in the FCA and PCA minutes have been corrected. With the completion of the FQR, the software is certified and accepted by the Air Force manager.

Summary

The DOD manager can ensure an orderly, well-planned development of a system by systematically completing each review and audit. This imposed organization can help the manager succeed in acquiring software which will identify and satisfy the allocated baseline. In addition, a systematic development reduces the number of design errors in a software package, thus reducing software maintenance and system life cycle costs.

APPENDIX B

GLOSSARY

This appendix defines the software engineering terms used in this thesis. The definitions are a subset of the IEEE Standard Glossary of Software Engineering Terminology except where otherwise indicated.

Definitions

acceptance testing: formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system.

algorithm: 1) a finite set of well-defined rules for the solution of a problem in a finite number of steps; for example, a complete specification of a sequence of arithmetic operations for evaluating sin x to a given precision (ISO), 2) a finite set of well-defined rules that gives a sequence of operations for performing a specific task.

audit: 1) an independent review for the purpose of assessing compliance with software requirements, specifications, baselines, standards, procedures, instructions, codes, and contractual and licensing requirements, 2) an activity to determine through investigation the adequacy of, and adherence to, established procedures, instructions, specifications, codes, and standards or other applicable contractual and licensing requirements, and the effectiveness of implementation.

baseline: 1) a specification or product that has been normally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures, 2) a configuration identification document or a set of such documents formally designated and fixed at a specific time during a configuration item's life cycle. Baselines, plus approved changes from those baselines, constitute the current configuration identification. For configuration management there are three baselines, as follows: a) functional baseline - the initial approved functional configuration, b) allocated baseline--the initial approved allocated configuration, and c) product baseline--the initial approved or conditionally approved product configuration identification.

<u>bottom-up</u>: pertaining to an approach that starts with the lowest level software components of a hierarchy and proceeds through progressively higher levels to the top level component; for example, bottom-up design, bottom-up programming, bottom-up testing. Contrast with top-down.

<u>code</u>: 1) a set of unambiguous rules specifying the manner in which data may be represented in a discrete form, 2) to represent data or a computer program in a symbolic form that can be accepted by a processor, 3) to write a routine, 4) loosely, one or more computer programs, or part of a computer program, 5) an encryption of data for security purposes.

<u>cohesion</u>: The degree to which the tasks performed by a single program module are functionally related. Contrast with coupling.

<u>computer</u>: 1) a functional unit that can perform substantial computation, including numerous arithmetic operations or logic operations, without intervention by a human operator during a run, 2) a functional programmable unit that consists of one or more associated processing units and peripheral equipment, that is controlled by internally stored programs, and that can perform substantial computation, including numerous arithmetic operations or logic operations, without human intervention.

<u>computer network</u>: a complex consisting of two or more interconnected computers.

<u>computer program</u>: a sequence of instructions suitable for processing by a computer. Processing may include the use of an assembler, a compiler, an interpreter, or a translator to prepare the program for execution as well as to execute it.

<u>computer sophisticate</u>: a person comfortable with the complexities of modern computer science. One able to cope with hardware and software which is not user-friendly. [author]

<u>computer system</u>: a functional unit, consisting of one or more computers and associated software, that uses

common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during their execution. A computer system may be a standalone unit or may consist of several interconnected units. Synonymous with ADP system, computing system.

configuration: 1) the arrangement of a computer system or network as defined by the nature, number, and the chief characteristics of the functional units. More specifically, the term configuration may refer to a hardware configuration or a software configuration, 2) the requirements, design, and implementation that define a particular version of a system or a system component, 3) the functional and/or physical characteristics of hardware/software as set forth in technical documentation and achieved in a product.

configuration audit: the process of verifying that all required configuration items have been produced, that the current version agrees with specified requirements, that the technical documentation completely and accurately describes the configuration items, and that all change requests have been resolved.

configuration item: 1) a collection of hardware or software elements treated as a unit for the purpose of configuration management, 2) an aggregation of hardware/software, or any of its discrete portions, that satisfies an end use function and is designated for configuration management. Configuration items may vary widely in complexity, size, and type from an aircraft, electronic or ship system to a test meter or round of ammunition. During development and initial production, configuration items are only those specification items that are referenced directly in a contract (or an equivalent in-house agreement). During the operation and maintenance period, any reparable item designated for separate procurement is a configuration item.

coupling: a measure of the interdependence among modules in a computer program. Contrast with cohesion.

Critical Design Review (CDR): a formal technical review of the design as depicted by the specification and flow diagrams, sufficiently detailed to enable the programmer to code and to assure that design requirements have been met before beginning coding. [29:77-79]

<u>data base</u>: 1) a set of data, part or the whole of another set of data, and consisting of at least one file that is sufficient for a given purpose or for a given data processing system, 2) a collection of data fundamental to a system, 3) a collection of data fundamental to an enterprise.

<u>error</u>: 1) a discrepancy between a computed, observed, or measured value or condition and the true, specific, or theoretically correct value or condition, 2) human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, incorrect translation, or omission of a requirement in the design specification. This is not a preferred usage.

<u>flowchart</u>: a graphical representation of the definition, analysis, or solution of a problem in which symbols are used to represent operations, data, flow, and equipment.

<u>Formal Qualification Review (FQR)</u>: the test, inspection, or analytical process by which products at the end stem or critical-item level are verified to have met specific procuring activity contractual performance requirements (specification or equivalent). [[illegible]7-79]

<u>Formal Qualification Tests (FQT)</u>: formal tests oriented toward testing of the functional and performance characteristics of the CPCI, normally using operationally configured equipment at the System DT&E site prior to the beginning of System DT&E. [29:77-79]

<u>Full-Scale Development Phase</u>: the period when the system/equipment and the principal items necessary for its support are designed, fabricated, tested, and evaluated. The intended output is, as a minimum, a preproduction system which closely approximates the final product, the documentation necessary to enter the production phase, and the test results which demonstrate that the production product will meet stated requirements (DODI 5000.1, AFR 800-2). [29:77-79]

Functional Configuration Audit (FCA): a formal audit to validate that the development of a CI has been completed satisfactorily and that the CI has achieved the performance and functional characteristics specified in the functional or allocated configuration identification. [29:77-79]

function decomposition: a method of designing a system by breaking it down into its components in such a way that the components correspond directly to system functions and subfunctions.

hardware: physical equipment used in data processing, as opposed to computer programs, procedures, rules, and associated documentation. Contrast with software.

high-level language: synonomous with higher order language.

higher order language: a programming language that usually includes features such as nested expressions, user defined data types, and parameter passing not normally found in lower order languages, that does not reflect the structure of any one given computer or class of computers, and that can be used to write machine independent source programs. A single, higher order language statement may represent multiple machine operations. Contrast with machine language, assembly language.

instruction: 1) a program statement that causes a computer to perform a particular operation or set of operations, 2) in a programming language, a meaningful expression that specifies one operation and identifies its operands, if any.

integration: the process of combining software elements, hardware elements, or both into an overall system.

interface: 1) a shared boundary. An interface might be a hardware component to link two devices or it might be a portion of storage or registers accessed by two or more computer programs, 2) to interact or communicate with another system component.

machine language: a representation of instructions and data that is directly executable by a computer. Contrast with assembly language, higher order language.

maintainability: 1) the ease with which software can be maintained, 2) the ease with which maintenance of a functional unit can be performed in accordance with prescribed requirements.

modular decomposition: a method of designing a system by breaking it down into modules.

modular programming: a technique for developing a system or program as a collection of modules.

modularity: the extent to which software is composed of discrete components such that a change to one component has minimal impact on other components.

module: 1) a program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from, and assembler, compiler, linkage editor, or executive routine, 2) a logically separable part of a program.

nest: 1) to incorporate a structure or structures of some kind into a structure of the same kind. For example, to nest one loop (the nested loop) within another loop (the nested subroutine) within another subroutine (the nesting subroutine), 2) to place subroutines or data in other subroutines or data at a different hierarchical level so that subroutines can be executed as recursive subroutines or so that the data can be accessed recursively.

network: 1) an interconnected or interrelated group of nodes, 2) in connection with a disciplinary or problem oriented qualifier, the combination of material, documentation, and human resources that are united by design to achieve certain objectives; for sample, a social science network, a science information network.

operating system: software that controls the execution of programs. An operating system may provide services such as resource allocation scheduling, input/output control, and data management. Although operating systems are predominantly software, partial or complete hardware implementations are possible. An operating system provides support in a single spot rather than forcing each program to be concerned with controlling hardware.

Physical Configuration Audit (PCA): a formal examination of the technical documentation (specification, handbooks, and manuals) to determine their compatibility with the qualified CPCI. [29:77-79].

Physical Design Review (PDR): A formal review of the preliminary design of a CI to 1) evaluate technical progress, 2) determine its compatibility with the requirements of the CI Development Specification, and 3) establish the existence and compatibility of the physical and functional interfaces among CI equipment, facilities, computer programs, and personnel. [29:77-79]

preliminary design: 1) the process of analyzing design alternatives and defining the software architecture. Preliminary design typically includes definition and structuring of computer program components and data, definition of the interfaces, and preparation of timing and sizing estimates, 2) the result of the preliminary design process.

program: 1) a computer program, 2) a schedule or plan that specifies actions to be taken, 3) to design, write, and test computer programs.

quality assurance: a planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements.

reliability: 1) the ability of an item to perform a required function under stated conditions for a stated period of time, 2) see software reliability.

software development plan: a project plan for the development of a software product. Synonymous with computer program development plan.

software documentation: technical data or information, including computer listings and printouts, in human-readable form, that describe or specify the design or details, explain the capabilities, or provide operating instructions for using the software to obtain desired results from a software system.

software engineering: the systematic approach to the development, operation, maintenance, and retirement of software.

software reliability: 1) the probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered, 2) the ability of a program to perform a required function under stated conditions for a stated period of time.

specification: 1) a document that prescribes, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or system component, 2) the process of developing a specification, 3) a concise statement of a set of requirements to be satisfied by a product, a material or process indicating, whenever appropriate, the procedure by means of which it may be determined whether the requirements given are satisfied.

structured design: a disciplined approach to software design that adheres to a specified set of rules based on principles such as top-down design.

structured program: a program constructed of a basic set of control structures, each one having one entry point and one exit. The set of control structures typically includes: sequence of two or more instructions, conditional selection of one of two or more instructions, conditional selection of one of two or more instructions or sequences of instructions, and repetition of an instruction or a sequence of instructions.

structured programming: 1) a well-defined software development technique that incorporates top-down design and implementation and strict use of structured program control constructs, 2) loosely, any technique for organizing and coding programs that reduces complexity, improves clarity, and facilitates debugging and modification.

System Design Review (SDR): a design review conducted to evaluate the optimization, correlation, completeness, and risk associated with the allocated technical requirements. [29:77-79]

System Requirements Review (SRR): a system engineering review to ascertain the adequacy of the contractor's efforts in defining system requirements. It will be conducted when a significant portion of the system functional requirements has been established. [29:77-79]

system software: software designed for a specific computer system or family of computer systems to facilitate the operation and maintenance of the computer system and associated programs, for example, operating systems, compilers, utilities.

test plan: a document prescribing the approach to be taken for intended testing activities. The plan typically identifies the items to be tested, the testing to be performed, test schedules, personnel requirements, reporting requirements, evaluation criteria, and any risks requiring contingency planning.

test repeatability an attribute of a test indicating whether the same results are produced each time the test is conducted.

testability: 1) the extent to which software facilitates both the establishment of test criteria and the evaluation of the software with respect to those criteria, 2) the extent to which the definition of requirements facilitates analysis of the requirements to establish test criteria.

testing: the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.

top-down: pertaining to an approach that starts with the highest level component of a hierarchy and proceeds through progressively lower levels; for example, top-down design.

top-down design: the process of designing a system by identifying its major components, decomposing them into their lower level components, and iterating until the desired level of detail is achieved. Contrast with bottom-up design.

user-friendly: easy to use or understand by the non-expert. [author]

validation: the process of evaluating software at the end of the software development process to ensure compliance with software requirements.

Validation Phase: the overall objective of the Validation Phase is to determine whether to proceed with Full-Scale Development. The ultimate goal of the V[illegible]on Phase, where development is to be performed by a contractor, is to establish firm and realistic performance specifications (Allocated Baseline), which meet the operational and support requirements. [29:77-79]

verification: 1) the process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase, 2) formal proof of program correctness, 3) the act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services, or documents conform to specified requirements.

walk-through: a review process in which a designer or programmer leads one or more other members of the development team through a segment of design or code that he or she has written, while the other members ask questions and make comments about technique, style, possible errors, violation of development standards, and other problems.

APPENDIX C

FORMAL PROGRAM DESIGN METHODOLOGIES

No single design methodology has been shown to be "correct" for all types of problems. Therefore, the following suggestions cover a variety of applications (2:13).

Functional Decomposition. Refers to:

> the divide-and-conquer technique applied to programming [as shown in Figure 10]. By viewing the stepwise decomposition of the problem and the simultaneous development and refinement of the program as a gradual progression to levels of greater and greater detail, functional decomposition [can be characterized] as a top-down approach to problem-solving. Conversely, groups of instructions sequences [can be formed and layered] together into "action clusters," starting at the atomic machine instruction level and working our way up to the complete solution. This approach leads to a bottom-up method [2:19].

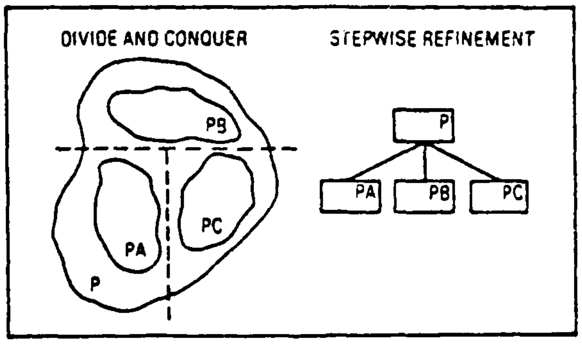


Figure 10. Functional Decomposition (2:19)

Data Flow Design. This method has also been known as "transform-centered design" and "composite design" (2:23).

In its simplest form, [data flow design] is nothing more than functional decomposition with respect to data flow. Each block of the structure chart is obtained by successive application of the engineering definition of a black box that transforms an input data stream into an output data stream. When these transforms are linked together appropriately, the computational process can be modified and implemented much like an assembly line that merges streams of input parts and outputs streams of final products [2:23].

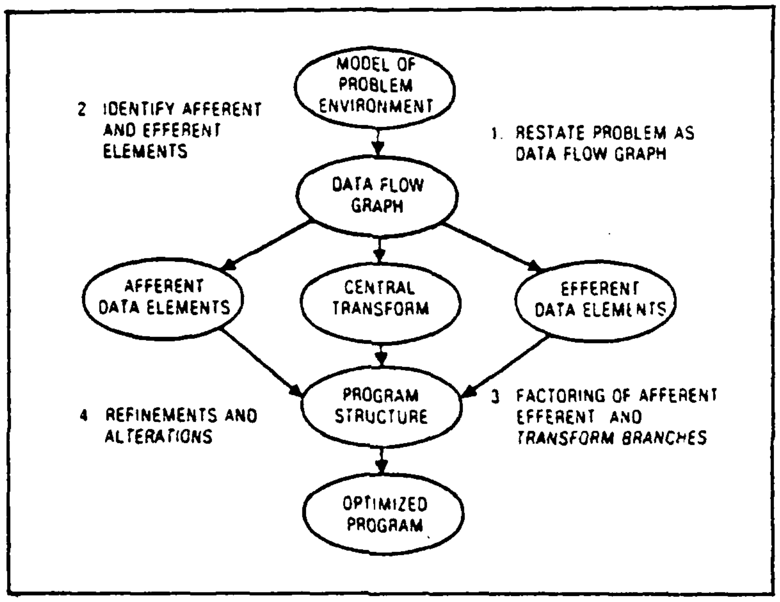

MODEL OF PROBLEM ENVIRONMENT
2 IDENTIFY AFFERENT AND EFFERENT ELEMENTS
1. RESTATE PROBLEM AS DATA FLOW GRAPH
DATA FLOW GRAPH
AFFERENT DATA ELEMENTS
CENTRAL TRANSFORM
EFFERENT DATA ELEMENTS
4 REFINEMENTS AND ALTERATIONS
PROGRAM STRUCTURE
3 FACTORING OF AFFERENT EFFERENT AND TRANSFORM BRANCHES
OPTIMIZED PROGRAM

Figure 11. Data Flow Design Procedure (2:24)

Data Structure Design. This design follows the hypothesis that:

A program views the world through its data structures and that, therefore, a correct model of the data structures can be transformed into a program that incorporates a correct model of the world. The importance of this view is emphasized by Michael Jackson's words that "a program that doesn't directly correspond to the problem environment is not poor, is not bad, but is wrong [2:26].

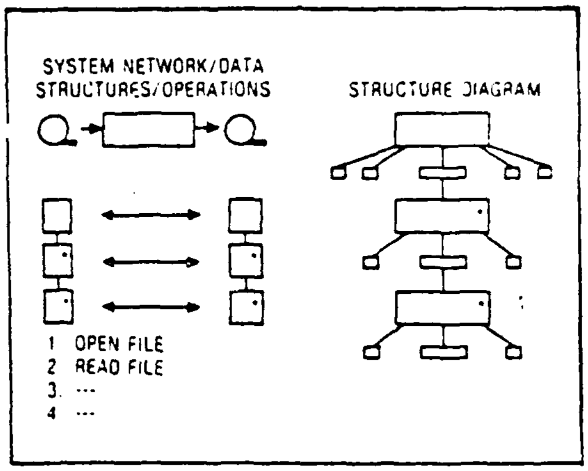


Figure 12. Data Structure Design Method (2:27)

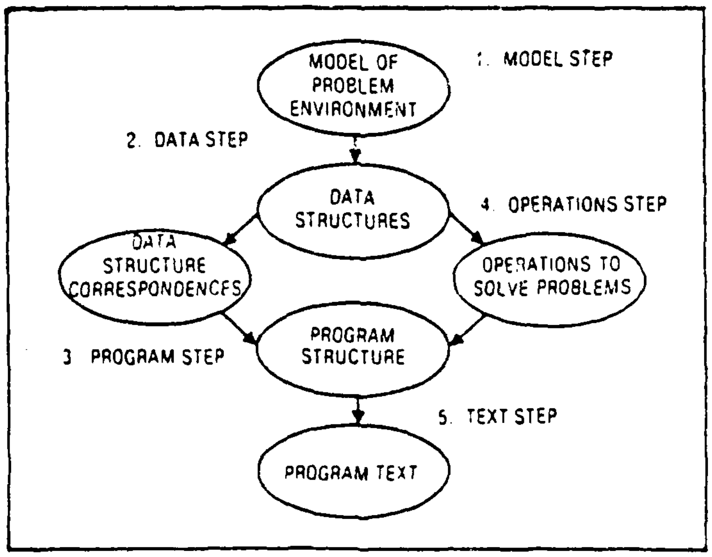


Figure 13. Basic Data Structure Design Procedure (2:27)

A Programming Calculus. Proving the "correctness" of a program is "disappointingly difficult [2:29]." This method uses a simultaneous process where the program and the proof are constructed together. Bergland discusses the design strategy as follows:

> The initial design task consists of formally specifying the required result as an assertion stated in the predicate calculus. Given this desired post-condition, one must derive and verify the appropriate pre-conditions while working back through the program being constructed. The program and even individual statements play a dual role in that they must be viewed in both an operational way and as predicate transformers. The method is a top-down method to the extent that both the resulting program and the predicates can be formed in stages by a sequence of stepwise refinements [2:29].

Methodology Comparison. Many claims have been made about the different strategies for software design, as seen in Figure 14.

> For functional decomposition, the proponents have largely said "D is [a] good design, believe me." For data flow design methods, people have said, "Program C is letter than program D. Let me tell you why." For data structure design methods, the claim is that "B is right, C and D are wrong. A program that works isn't necessarily right." In the programming calculus, the contention is that "Program A is probably correct. B, C, and D are unproven [2:35].

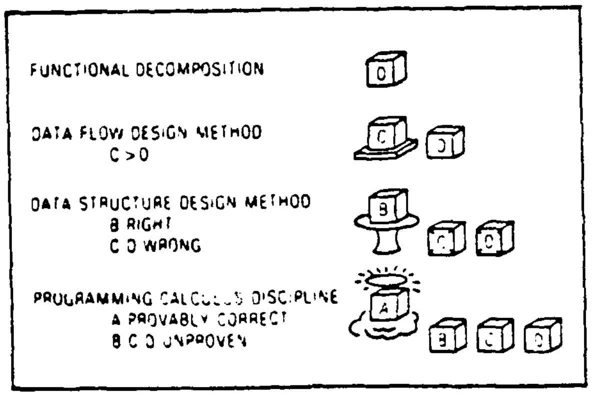


Figure 14. Summary of Program Design Methodology Claims (2:35)

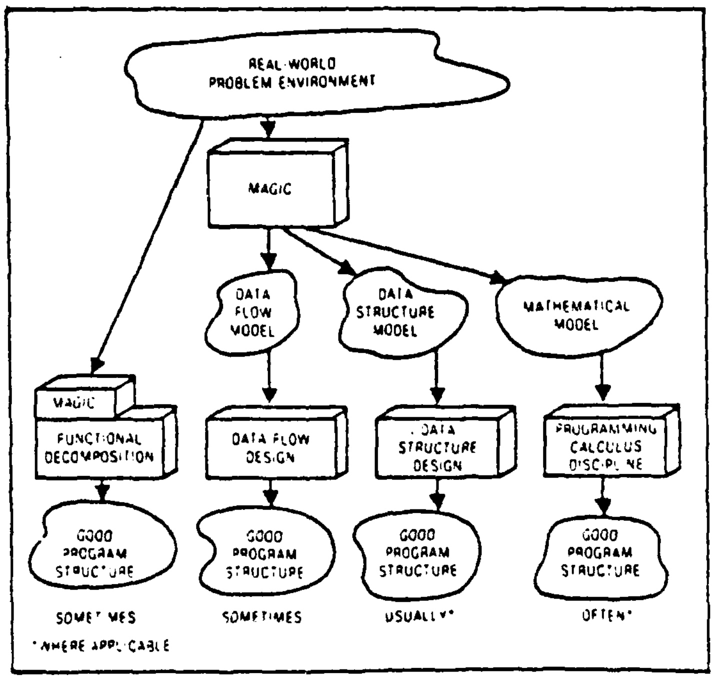


Figure 15. Current State-of-the-Art (2:35)

Other Design Recommendations. G. D. Bergland identified four specific design methodologies. These additional recommendations are more suitable for the software procurement novice.

Top-down design: In the top-down design approach, the system is approached as a whole from the outside. The system is then subdivided into a limited number of modules with specified interfaces. Each module is then subdivided into submodules (26).

Factoring: Like top-down design, this strategy breaks the problem into smaller, more manageable portions. Some guidelines include:

a. reduce the module size so that the entire module fits on one page.

b. clarify the system by modularizing.

c. avoid redundant functions.

d. separate work things (calculating and editing) from management things (decisions or module calls).

e. create more useful modules by ensuring each module does just one function (thereby being universally useful.)

Avoid Decision Splits: This refers to a situation where the data required for the recognition part of the decision is in a different module from the decision itself. This practice is difficult to implement but should be attempted (26).

Maintain Local Error Reporting: Report an error from the module that detects the error and knows what it is. This helps locate the source of the error, is easier to update the error message, avoids duplication and helps keep wording and format consistent.

Avoid State Memory: This refers to data internal to a module that survives unchanged from one use of the module to the next. State memory causes problems in this situation because its effects are unpredictable. It also makes the module harder (thus more expensive) to maintain (26).

These recommendations have been provided because they have gained notoriety as some of the better precautions to reduce software maintenance. The list of available "precautions" is surely unending. Nevertheless, these recommendations will contribute significantly to the reduction of software life-cycle costs.

APPENDIX D

MEASURES OF GOOD DESIGN

Among the primary measures of good design are module relatedness measures. They explain how portions of computer code (modules) are connected. Two module relatedness measures are coupling and cohesion.

Coupling

Coupling measures intermodule strength; the degree of interdependence between two modules. Coupling should be minimized to make modules as independent as possible (31; 26). The links on a chain have maximum coupling. If one link fails the other links become useless. Conversely, if the modules of a software system have minimum coupling, failure of one module will have minimum effect on the system. Minimized or loosest possible coupling is desired because:

1. The fewer connections there are between two modules, the less the chance there is for a bug in one module to appear as a symptom in another.

2. We can change the software in a module with minimum risk of having to change other modules.

3. While maintaining a module we don't want to worry about the coding details of other modules [31:102-103].

Six different types of module coupling may occur between modules. In order of the loosest or best to tightest or worst, they are:

Data: Modules communicate by by sending only those data elements required (26). Each element (or parameter) is either a single field or a table in which each entry contains the same type of information (31:103).

Stamp: Modules communication refers to the same data structure (26; 31:103; 34:43). The structure contains fields which are not required and are excess data. Therefore, when an element of data is manipulated in the structure it affects all the modules which refer to the structure. It will even affect those modules which do not interact with the actual data being serviced.

Control: Modules are control coupled if one module passes to a second module a piece of information intended to control the internal logic of the second module (31:103).

External: Modules refer to an externally declared data element [34:43].

Common: Modules refer to the same global data area (26).

Content: One module refers to data inside of another, or if one module alters a statement in another (31:113; 34:43; 26).

Two modules may have more than one type of coupling, or have the same type of coupling several times (31:103). Minimizing coupling will reduce the time and expense required for software maintenance during the life of the system. Another measure of the design of a system is the strength within each module, or its cohesion.

Cohesion

Cohesion refers to the intramodule strength. It "is the measure of the strength of functional association of elements within a module [31:118]." The 'elements' are the computer instructions. The preference is for modules whose elements are strongly and genuinely related to one another. Ideally, a module contains elements that all contribute to the execution of one problem-related task. An example of good cohesion would be a module which calculates the area of a circle given the radius. It has one purpose. An example of bad cohesion is a module whose elements contribute to activities with no meaningful relationship to one another (31:129). Cohesion is a good measure of a module's maintainability. A module with good cohesion is easier (thus less expensive) to maintain than a module with bad cohesion.

The seven levels of cohesion in order of their maintainability (from best to worst) are:

Functional: This most desired form of cohesion is characterized by modules that perform a single specific function (34:43). All the elements in the module contribute to the execution of one, and only one, problem related task (31-119).

Sequential: In this form of cohesion module, action consists of several logical functions operating on data in a predetermined order (34:43). The output of one function (or operation) is the input to the next (26; 31:125).

Communicational: Software with this intramodule strength uses the same input, but the output is not related. The order of the operations is not important (26; 31:125).

Procedural: "A procedurally cohesive module is one whose elements are involved in different and possibly unrelated activities, in which control flows from each activity to the next [31:124]." A specified order exists, but there is no relation between steps, and control flows from one element to another (26).

Temporal: The elements of this module are involved in activities that are related in time (31:125). A key characteristic of this form of intramodule strength is 'initialization' (26; 34:43).

Logical: These elements contribute to activities of the same general category. They are substitute choices for each other and generally are represented by a decision on a structure chart. It is used to overlap parts of functions which have the same lines of code (26; 31:125).

Coincidental: The elements in this module contribute to activities with no meaningful relationship to one another (31:129). It is a 'coincidence' that all functions are in the same module (26).

> Cohesion is a measure of module strength, and acts somewhat like a chain, holding together the activities in the module. If all the activities of a module are related by more than one level of cohesion, the module has the strength of the strongest level of cohesion; this is the chains-in-parallel rule [31:133].

APPENDIX E

STRUCTURED PROGRAMMING

Structured programming is the formulation of programs as hierarchical nested structures of statements and objects of computation (49). The goal of structured programming is to reduce program complexity, and improve program clarity. These two attributes will help make the software program be more 'user friendly' and have lower software maintenance requirements. Structured programming reduces complexity by using only three control structures: 1) sequence; 2) conditional (if-then-else); and 3) loop (do-while). Figures 16 and 17 graphically depict the three control structures of structured programming.

Code designed using only these control structures is easier to read and understand. Code that is easy to read and understand is less difficult to update. Therefore, the use of structured programming can reduce software maintenance and computer system life-cycle costs.

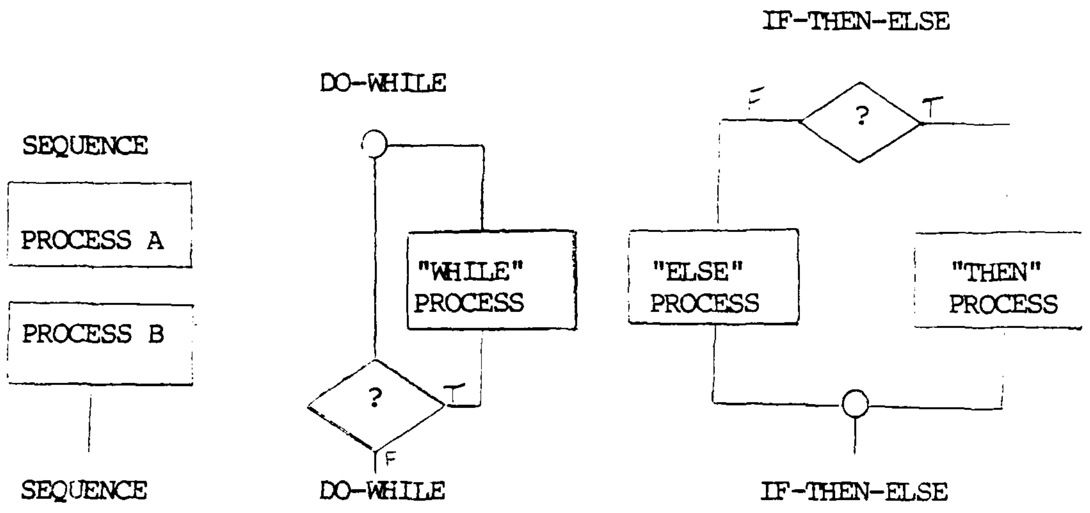


Figure 16. Basic Control Structures of Structured Programming (34:15)

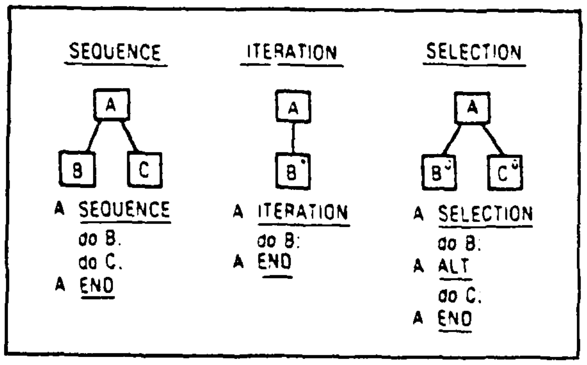


Figure 17. The Three Basic Control Flow Constructs (2:15)

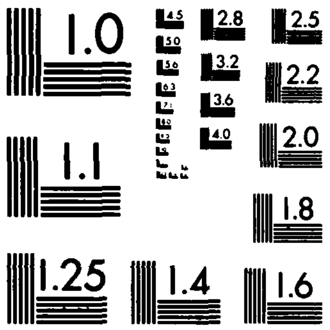
APPENDIX F

PROGRAMMING STYLE

AD-A134 363 MICROCOMPUTER SOFTWARE SYSTEM DEVELOPMENT: SUGGESTED REVISIONS TO MIL-STD..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF SYST.. V M HELBLING 2/2

UNCLASSIFIED SEP 83 AFIT-LSSR-10-83 F/G 5/1 NL

END


1.0
4.5
2.8
2.5
5.0
3.2
2.2
5.6
6.3
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

In the last twenty years, the concept of programming style has been changed drastically.

> If you asked someone what good programming style was, you would likely have received (if you didn't get a blank stare) a lecture on
>
> 1. how to save microseconds
> 2. how to save words of memory
> 3. how to draw neat flowcharts
> 4. how many comments to write per line of code [21:303].

Today, some microcomputer programmers are primarily concerned with 'efficiency' and 'clever' use of machine capabilities. These practices are dangerous and result in software that is difficult (therefore expensive) to update. Kernighan and Plauger recommend these elements of programming style:

<u>Expression</u>: At the lowest level of coding, individual statements and small groups of statements have to be expressed so they read clearly (21:304).

<u>Structure</u>: The larger structure of the code should also read clearly. It should be written with only a handful of control-flow primitives (such as if-then-else). This is one aspect of what we mean by structured programming (see Appendix E). Coding in this set of well-behaved structures makes code readable, thus more understandable and likely to be right. This reduces software maintenance cost by making the code easier to change and debug (21:304).

Robustness: A program should work not just on the easy cases, or on the well-exercised ones, but all the time (21:304).

Efficiency and Instrumentation: Until we have a working piece of code, we don't always know where it spends its time. And until we know that, talk of changing it "for efficiency" is foolish. Write the whole program clearly. If it is then too slow or too big, change to a better algorithm (21:305).

Documentation: If you write the code with care in the detailed expression, using the fundamental structures, and choosing a good representation, most of the code will be self-documenting--it can be read (21:305).

Kernighan and Plauger did not emphasize a need for the use of flowcharts. In fact, studies by Schneiderman, Mayer, McKay, and Heller conject that "flowcharts may even be a hindrance [34:65]." Kernighan and Plauger did recommend these guidelines for expression:

1. Don't be too clever.
2. Don't be too complicated.
3. Be clear versus efficient.
4. Don't hesitate to rewrite the program. (21:305-309).

Milne's guidelines parallel those suggestions:

1. Don't be too clever or complicated.
2. Don't optimize the code.
3. Stick to basic control structures (Appendix E).
4. Make sure every comment means something.
   Don't have comments echo the code.
5. Use an appropriate language, but be clear in the language you use.
6. Use descriptive names for objects.
7. Avoid negative logic.
8. Don't be unnecessarily complicated.
9. Avoid temporary variables (26).

The DOD manager should request that the contractor follow these guidelines. Programmers often claim that they don't have time to worry about niceties like style (21:309). The DOD manager must resist abdicating to this complaint. The time saved when problems are encountered in testing and when software modifications are performed on complicated, unclear software will far outweigh any savings during the software development.

SELECTED BIBLIOGRAPHY

A. REFERENCES CITED

1. Babel, Philip S. ASD Computer Resource Focal Point, Deputy for Engineering, ASD/EN, Wright-Patterson AFB OH. Personal Interview. 23 March 1983.

2. Bergland, G. D. "A Guided Tour of Program Design Methodologies," Computer, October 1981, pp. 13-37.

3. Brooks, Frederick P., Jr. The Mythical Man-Month. Phillipines: Addison-Wesley Publishing Company, Inc., 1975.

4. Bunyard, Major General Jerry Max, USAF, and James Mike Coward. "Today's Risks in Software Development: Can They be Significantly Reduced?" Concepts, Vol. 5, No. 4 (Autumn 1982), pp. 79-84.

5. Caine, Stephen H., and E. Kent Gordon. "PDL - A Tool for Software Design," Proceedings, National Computer Conference, AFIPS Press, 1975.

6. Clapp, Judith A. "Designing Software for Maintainability," Computer Design, September 1981, pp. 197-204.

7. Clark, Lieutenant General Lynnwood E., USAF. Commander, Alaskan Air Command, Elmendorf AFB AK. Personal interview. 17 June 1983.

8. Craddoc, Doc. Manager for Software Quality, Texas Instruments, Austin TX. Telephone interview. 19 July 1983.

9. Dayton Daily News. "Sally Ride: No Trade for Space Trip," Vol. 106, No. 338 (14 August 1983), p. 2.

10. Douglas, Captain Frank E., III, USAF. "Physical Configuration Audit of Embedded Computer Programs." IEEE 1979 National Aerospace and Electronics Conference. New York: IEEE, 1979.

11. Dunn, Robert H., and Richard S. Ullman. "A Workable Software Quality/Reliability Plan," IEEE Proceedings 1978: Annual Reliability and Maintainability Symposium. New York: IEEE, 1978.

12. Ferris, Michael. "Automating America's Air Defense: A Network of XT's Stands Ready," Softalk for the IBM Personal Computer, Vol. 2 (June 1983), pp. 75-81.

13. Gaudreau, Captain Skip, USAF. Headquarters Alaskan Air Command, Elmendorf AFB AL. Personal interview. 16 June 1983.

14. Greentree Computer Corporation. How to Choose Micro Computer Software. Rockville MD: Greentree Computer Corporation, 1983.

15. Grove, Mark H. "DOD Policy for Acquisition of Embedded Computer Resources," Concepts, Vol. 5, No. 4 (Autumn 1982), pp. 9-36.

16. Henderson, John. Systems Engineer, International Business Machines, Dayton OH. Telephone interview. 28 July 1983.

17. Higgins, David A. "Structured Programming with Warnier-Orr Diagrams," Byte, December 1977, pp. 104-176.

18. Howard, Lieutenant Colonel Dennis D. Chief of Data Automation, Elmendorf AFB AL. Personal interview. 17 June 1983.

19. Huang, J. C. "Program Instrumentation and Software Testing," Computer, April 1978.

20. Huskey, Major Charlie D. Lessons Learned in the Use of Microcomputers in Systems Development. Alexandria VA, Defense Technical Information Center, 1980.

21. Kernighan, Brian W., and P. J. Plauger. "Programming Style: Examples and Counterexamples," Computing Surveys, Vol. 6, No. 4 (December 1974), pp. 303-319.

22. Lang, Walter N. "What the Computer has Wrought," Air Force, Vol. 66, No. 7 (July 1983), pp. 68-72.

23. Marriot, Phil. Manager of Software Quality, National Cash Register Corporation, Dayton OH. Telephone interview. 21 July 1983.

24. Mathenia, Lynn. Operations Manager, The International Software Data Base, Ft. Collins CO. Telephone interview. 2 July 1983.

25. McCracken, Michael L. Former Chief System Test, 4602d Computer Services Squadron, Lowry AFB CO. Personal interview. 12 July 1983.

26. Milne, Rob. 1st Lieutenant USA. Professor, Department of Electrical Engineering, AFIT/ENG, Wright-Patterson AFB OH. Course EE 593. "Software Engineering," Spring Quarter 1983. Lectures. 18 March 1983 through 3 June 1983.

27. Morgan, D. E., and D. J. Taylor. "A Survey of Methods of Achieving Reliable Software," Computer, February 1977, pp. 44-52.

28. Myers, L. F. District Staff Manager, Data Base Administrator, Northwestern Bell, Omaha NE. Personal interview. 22 December 1982.

29. Neil, George. Software Acquisition Management Guidebook: Reviews and Audits. Alexandria VA: Defense Technical Information Center, 1977.

30. Nelson, Captain William, USAF. Software Quality Control, Hanscom AFB MA. Telephone interview. 16 February 1983.

31. Page-Jones, Meilir. The Practical Guide to Structured Systems Design. New York: Yourdon Press, 1980.

32. Peters, Lawrence J. Software Design: Methods and Techniques. New York: The Yourdon Press, 1981.

33. ________. "Software Representation and Composition Technique," Proceedings of the IEEE, Vol. 68, No. 9 (September 1980), pp. 1085-1093.

34. Pilcher, Major Russell Dean, USAF. "Techniques Available for Improving the Maintainability of DOD Weapon System Software," Unpublished master's thesis, Naval Postgraduate School, Monterey CA, June 1980.

35. Rasch, Major Ronald H., USAF. Associate Professor of Accounting and Information Systems, Air Force Institute of Technology, Wright-Patterson AFB OH. Personal interview. 29 June 1983.

36. Rock, Dick. Data Automation Manager, Union Pacific Railroad, Omaha NE. Personal interview. 23 December 1982.

37. Ross, Douglas T. "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1 (January 1977), pp. 16-34.

38. Rowell, Captain Phillip V., USAF. "Specifying Users' Requirements inthe Context of Military Intelligence Related Computer Systems." Unpublished master's thesis. LSSR 53-81, AFIT/LS, Wright-Patterson AFB OH, 19 September 1981. AD A113017.

39. Rubey, Raymond J. Professor, Department of Electrical Engineering, AFIT/ENG, Wright-Patterson AFB OH. Course EE 5.45, "Software Acquisition," Spring Quarter 1983. Lectures. 28 March 1983 through 3 June 1983.

40. Scharer, Laura. "Pinpointing Requirements," Datamation, April 1981, pp. 139-151.

41. Schiffres, Manuel. "Behind the Shakeup in Personal Computers," U.S. News and World Report, Vol. 94, No. 25 (27 June 1983), pp. 59-60.

42. Stanley, Phillip M. "A Design Approach to the Audit of Computer Information Systems," Information Processing 80. New South Wales Australia: North Holland Publishing Company, 1980.

43. Stay, J. F. "Hipo and Integrated Program Design," IBM Systems Journal, Vol. 15, No. 2 (1976), pp. 143-154.

44. Steininger, Partner Henry J., Arthur Young and Company. The Impact of Low Cost Computing Technologies on the Department of Defense. Washington DC: Arthur Young and Company, 10 April 1983.

45. Sumner, Eric E. Vice President, Computer Technologies and Military Systems, Bell Laboratories, Murray Hill NJ. Personal interview. 13 January 1983.

46. Trauboth, H. Software Testing and Validation Techniques for Highly Reliable Process-Information Systems. Marlow Buckinghamshire England: Vlasak and Company Limited, 1980.

47. U.S. Department of Defense. Technical Reviews and Audits for Systems, Equipments and Computer Programs. MIL-STD-1521A (USAF). Washington: Government Printing Office, 1 September 1972.

48. Walker, Dr. Michael G. "Auditing Software Development Projects: A Control Mechanism for the Digital Systems Development Methodology." IEEE 1979 COMPCON, Spring. New York: IEEE, 1979.

49. Wirth, N. "On the Composition of Well-Structured Programs," ACM Computing Surveys, December 1974.

50. Wolfe, Major H. Wayne, USAF. "Those Damned Computers," Air University Review, Vol. XXXIV, No. 4 (May-June 1983), pp. 48-55.

51. Zelkowitz, Marvin V. "Perspectives on Software Engineering," Computing Surveys, Vol. 10, No. 2 (June 1983), pp. 197-216.

52. Zonars, Demetries. Computer Center Director, ASD/EN, Wright-Patterson AFB OH. Telephone interview. 16 August 1983.

B. RELATED SOURCES

Air Force Systems Command. A Guide for Program Management. AFSCP 800-3. Wright-Patterson AFB OH, 9 April 1976.

Bersoff, Edward, Vilas D. Henderson, and Stan Sugel. "Software Configuration Management: A Tutorial," Computer, January 1979, pp. 97-115.

Boehm, Dr. B. W. "Software Engineering," Classics in Software Engineering. New York: Yourdon Press, 1978.

________. Software Engineering Economics. Englewood Cliffs NJ: Prentice-Hall, Inc., 1981.

Dlutsch, Michael S. "Software Project Verification and Validation," Computer, April 1981, pp. 54-70.

Fagan, M. E. "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, Vol. 15, No. 3 (1976).

Howden, William E. "A Survey of Static Analysis Methods," IEEE Tutorial: Software Testing and Validation Techniques. New York: IEEE, 1978.

Jensen, Randall W. "Structured Programming," Computer, March 1981, pp. 31-48.

________, and Charles C. Jones. Software Engineering. Englewood Cliffs NJ: Prentice-Hall, Inc., 1979.

Lehman, John H. "How Software Projects Are Really Managed," Datamation, January 1979, pp. 119-129.

Metzger, Phillip W. Managing a Programming Project. Englewood Cliffs NJ: Prentice-Hall, Inc., 1973.

Tzudeker, Harvey. "Software Configuration Management Testability and Traceability," Defense Systems Management Review, Vol. I, No. 6, pp. 106-115.

U.S. Department of Defense. Configuration Control: Engineering Changes, Deviations and Waivers. MIL-STD-480A. Washington: Government Printing Office, 29 December 1978.

________. Configuration Management: Practives for Systems, Equipment , Munitions, and Computer Programs. MIL-STD-483 (USAF). Washington: Government Printing Office, 21 March 1979.

________. Engineering Management. MIL-STD-499A (USAF). Washington: Government Printing Office, 1 June 1976.

________. Specification Practices. MIL-STD-490. Washington: Government Printing Office, 18 May 1972.